W3School 移动端教程合集

wizardforcel

Published with GitBook



目錄

W3School 移动端教程合集

来源：菜鸟教程

整理：飞龙

感谢菜鸟教程站长的翻译和奉献。

W3School Swift 教程

作者：W3School

来源：Swift 教程

Swift 简介



Swift 是一种支持多编程范式和编译式的开源编程语言,苹果于2014年WWDC（苹果开发者大会）发布，用于开发 iOS，OS X 和 watchOS应用程序。

Swift 结合了 C 和 Objective-C 的优点并且不受 C 兼容性的限制。

Swift在 Mac OS 和 iOS 平台可以和Object-C使用相同的运行环境。这意味着Swift程序可以运行于目前已存在的平台之上，包含 iOS 6 和 OS X 10.8 都可以运行Swift的程序。

更重要的, Swift 和 Obj-C 的代码可并存于单一程序内, 这种延伸就如同 C 和 C++ 的关系一样。

2015年6月8日，苹果于WWDC 2015上宣布，Swift将开放源代码，包括编译器和标准库。

谁适合阅读本教程？

本教程适合想从事移动端(iphone)开发或 OS X 应用的编程人员，如果之前有编程基础更好。

本教程所有实例基于 Xcode7.1（Swift 2.x 的语法格式） 开发测试。

第一个 Swift 程序

第一个 Swift 程序当然从输出 "Hello, World!" 开始，代码如下所示：

```
/* 我的第一个 Swift 程序 */
var myString = "Hello, World!"

print(myString)
```

Swift 环境搭建

Swift是一门开源的编程语言，该语言用于开发OS X和iOS应用程序。

在正式开发应用程序前，我们需要搭建Swift开发环境，以便更好友好的使用各种开发工具和语言进行快速应用开发。由于Swift开发环境需要在OS X系统中运行，因此其环境的搭建将不同于Windows环境，下面就一起来学习一下swift开发环境的搭建方法。

成功搭建swift开发环境的前题：

1. 必须拥有一台苹果电脑。因为集成开发环境XCode只能运行在OS X系统上。
2. 电脑系统必须在OS 10.9.3及以上。
3. 电脑必须安装Xcode集成开发环境。

Swift 开发工具Xcode下载

Swift 开发工具官网地址：https://developer.apple.com/xcode/download/。

Swift 开发工具百度软件中心下载（国内比较快）：http://rj.baidu.com/soft/detail/40233.html

Swift 源代码下载：https://swift.org/download/#latest-development-snapshots

下载完成后，双击下载的 dmg 文件安装，安装完成后我们将 Xcode 图标踢移动到应用文件夹。

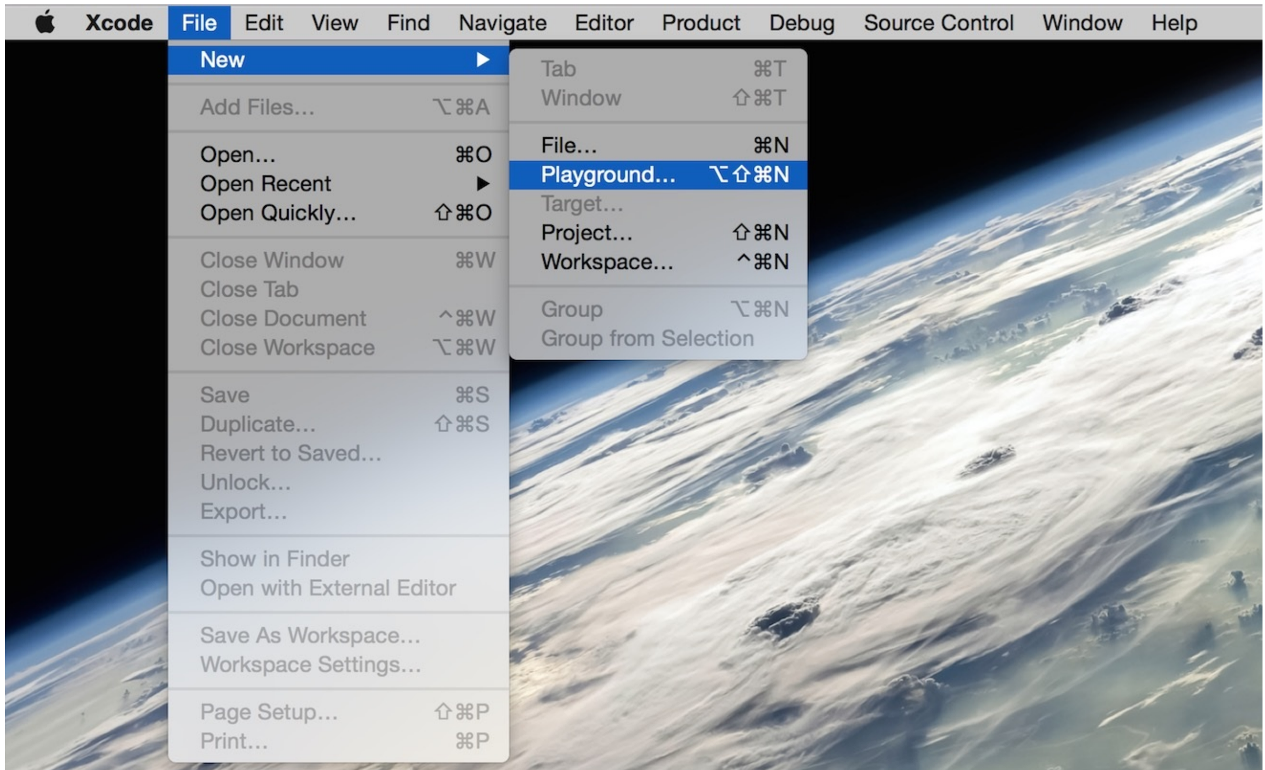


你也可以在 App Store 中搜索 xcode 安装，如下图所示：

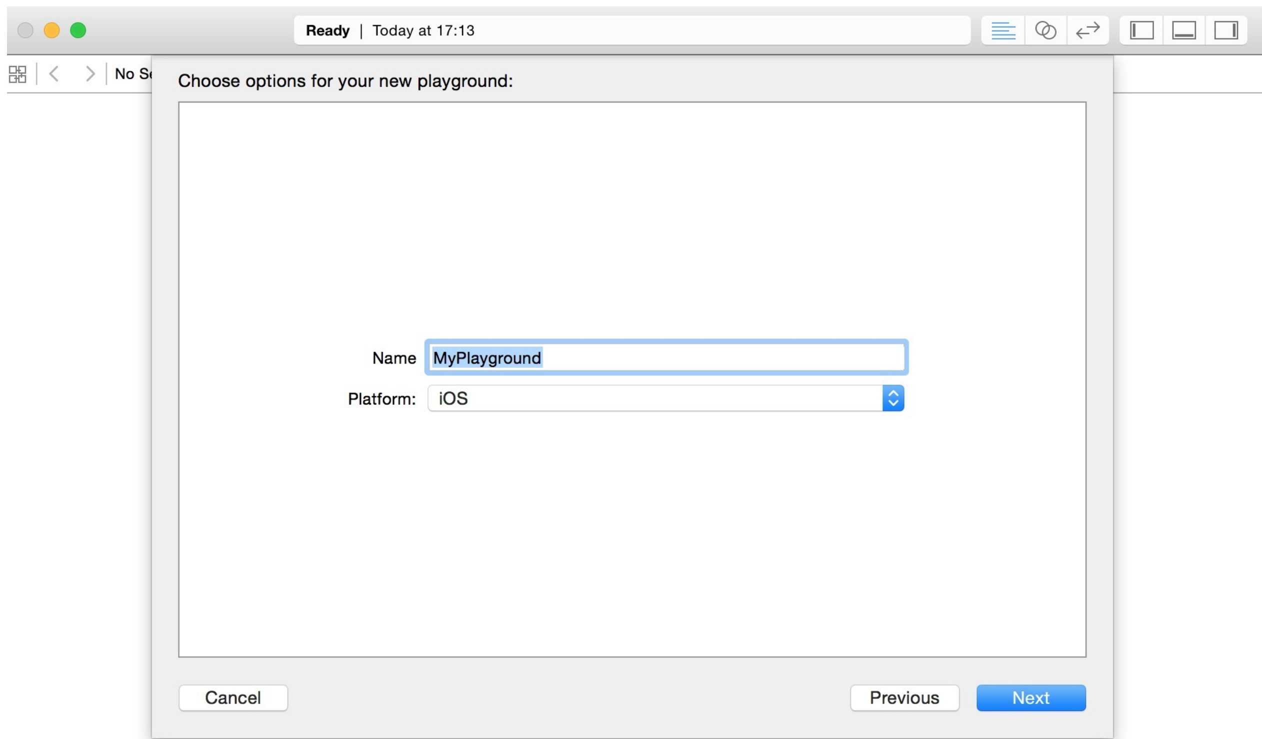
第一个 Swift 程序

Xcode 安装完成后，我们就可以开始编写 Swift 代码了。

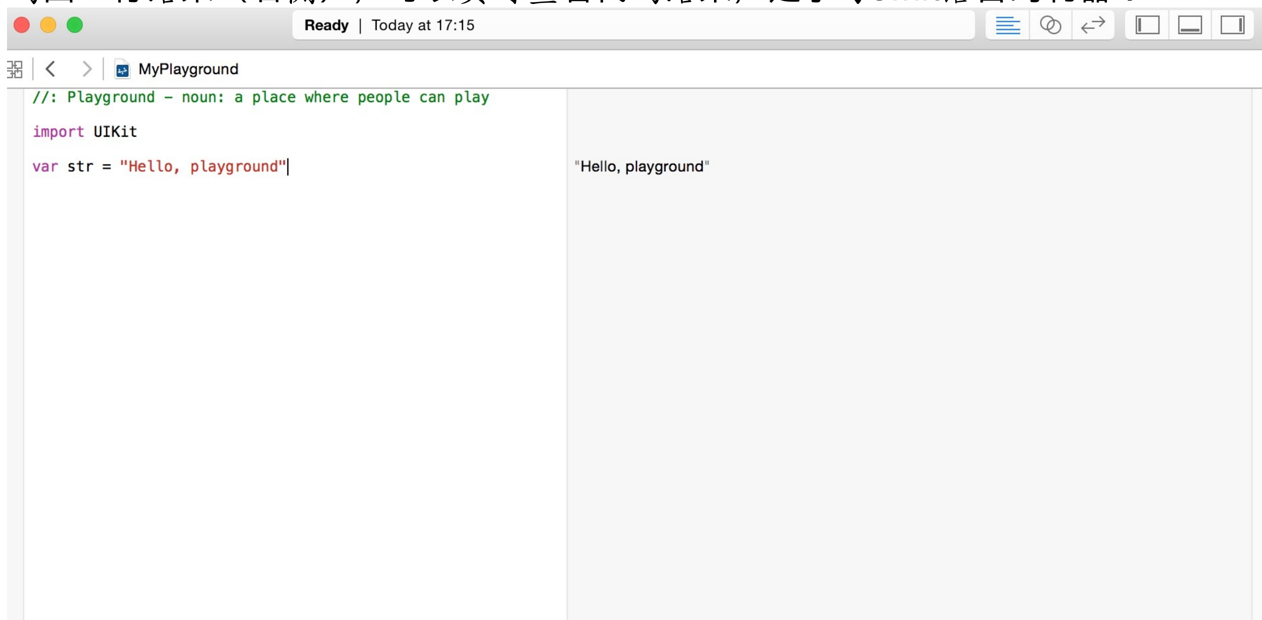
接下来我们在应用文件夹打开 Xcode，打开后在屏幕顶部选择 File => New => Playground。



接着 为 playground 设置一个名字并选择 iOS 平台。



Swift 的 playground 就像是一个可交互的文档，它是用来练手学swift的，写一句代码出一行结果（右侧），可以实时查看代码结果，是学习swift语言的利器！



以下是 Swift Playground 窗口默认的代码：

```
import UIKit

var str = "Hello, playground"
```

如果你想创建 OS x 程序，需要导入 Cocoa 包 **import Cocoa** 代码如下所示：

```
import Cocoa

var str = "Hello, playground"
```

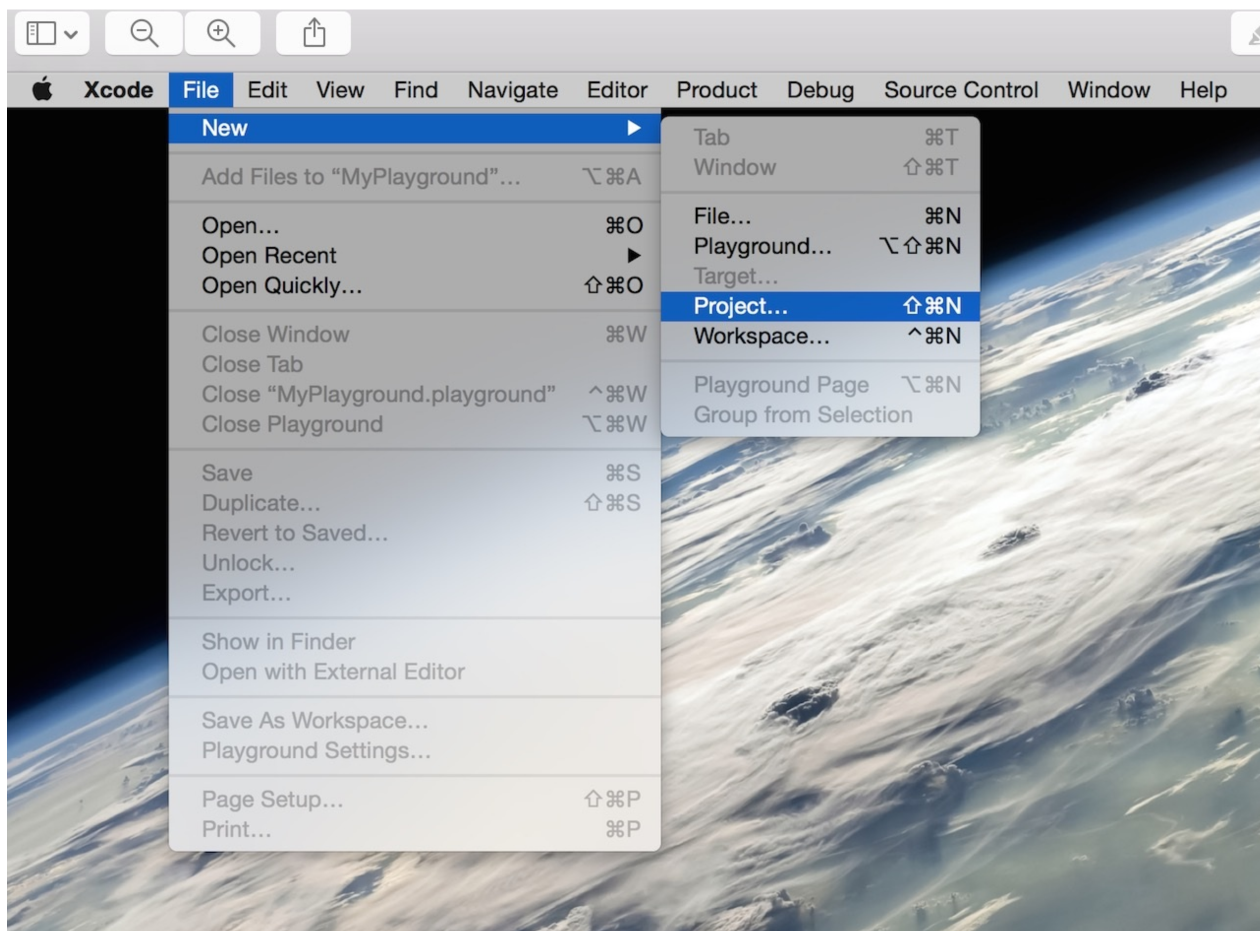
以上程序载入后，会在Playground 窗口右侧显示程序执行结果：

```
Hello, playground
```

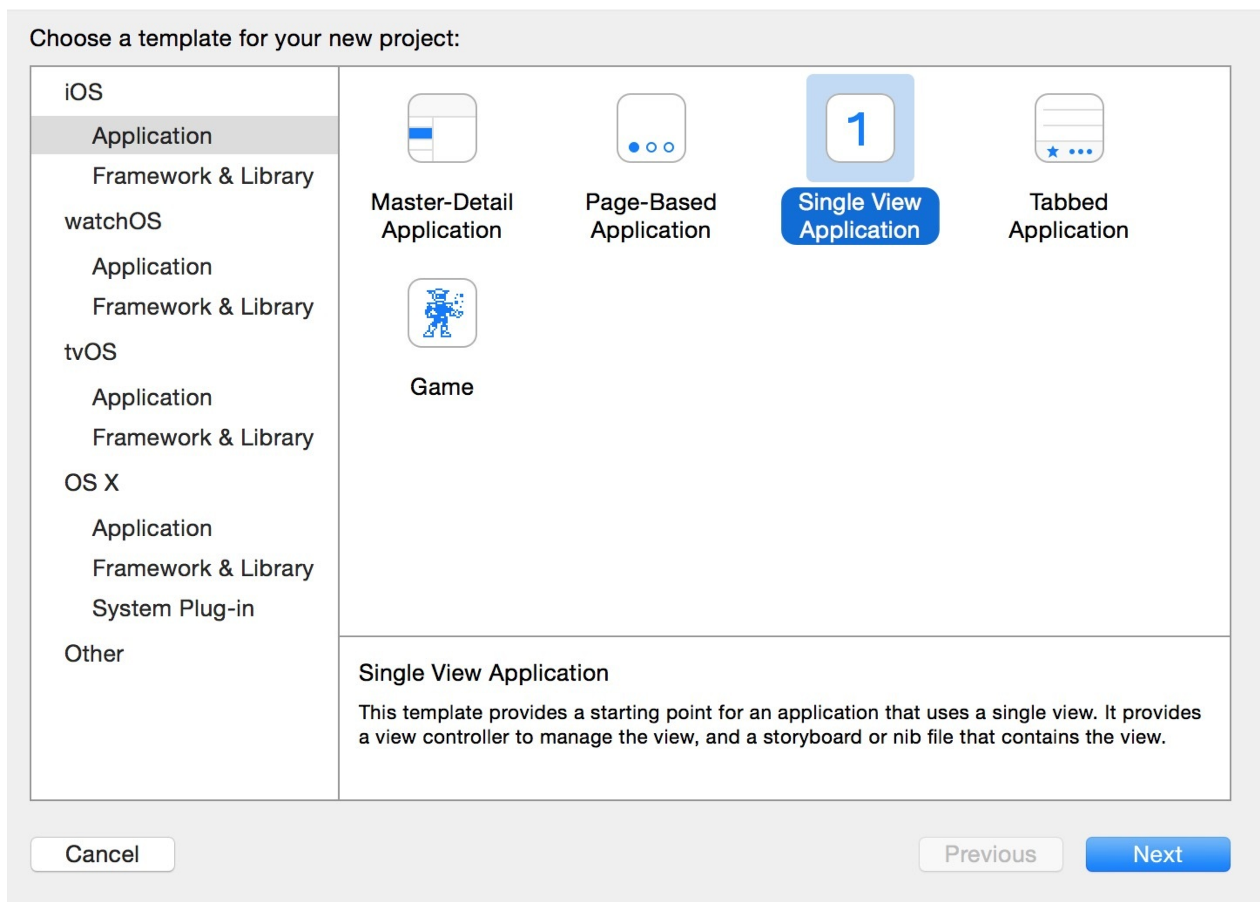
至此，你已经完成了第一个 Swift 程序的学习，恭喜你入门了。

创建第一个项目

1、打开 xcode 工具，选择 File => New => Project



2、我们选择一个"Single View Application"，并点击"next"，创建一个简单示例app 应用。



3、接着我们输入项目名称（ProductName),公司名称（Organization Name),公司标识前缀名（Organization identifier) 还要选择开发语言(Language),选择设备（Devices)。

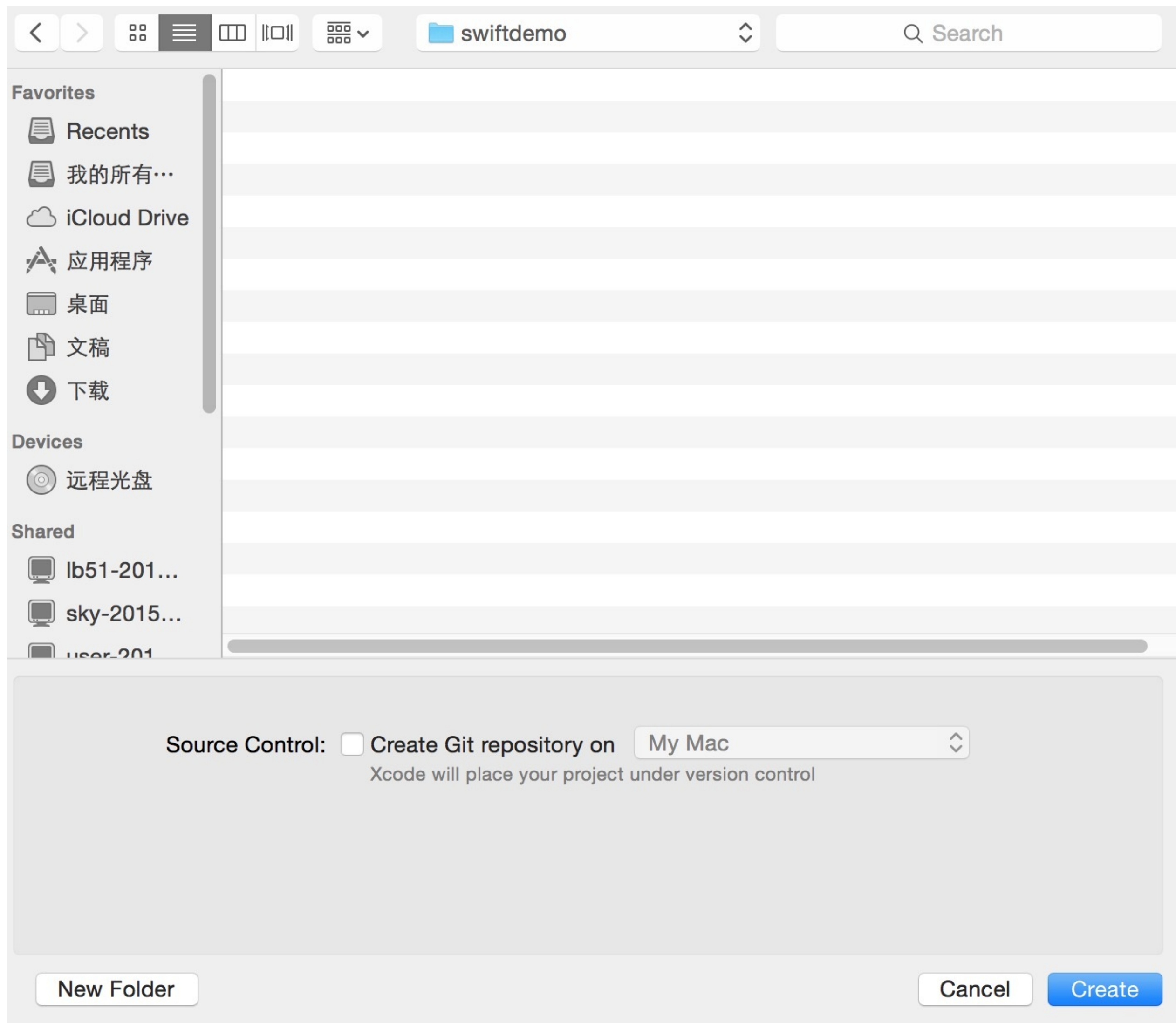
其中Language有两个选项：Objective-c和swift，因为我们是学习swift当然选择swift项了。 点击"Next"下一步。

Choose options for your new project:

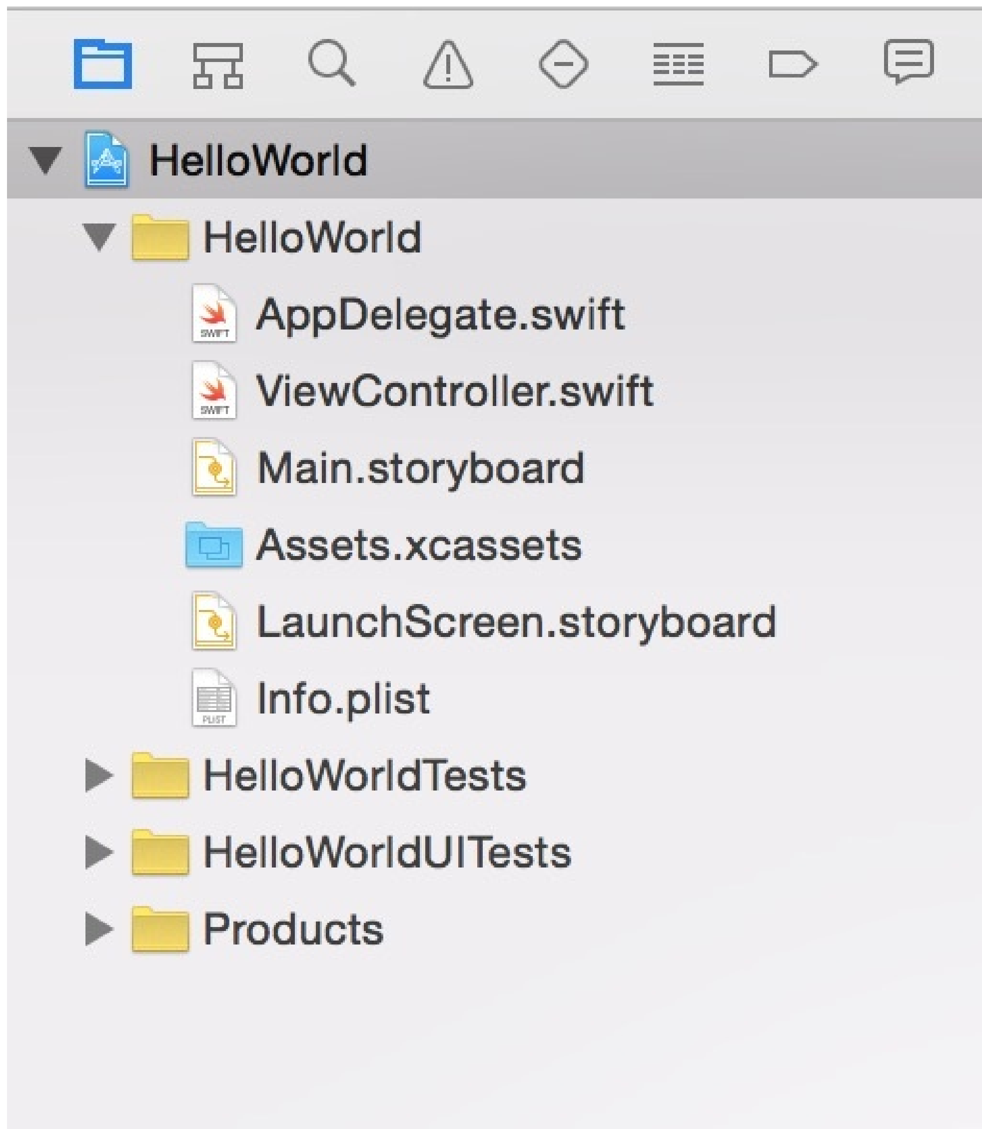
Product Name: HelloWorld
Organization Name: 菜鸟教程
Organization Identifier: com.runoob
Bundle Identifier: com.runoob.HelloWorld
Language: Swift
Devices: iPhone
Use Core Data
Include Unit Tests
Include UI Tests

Cancel
Previous
Next

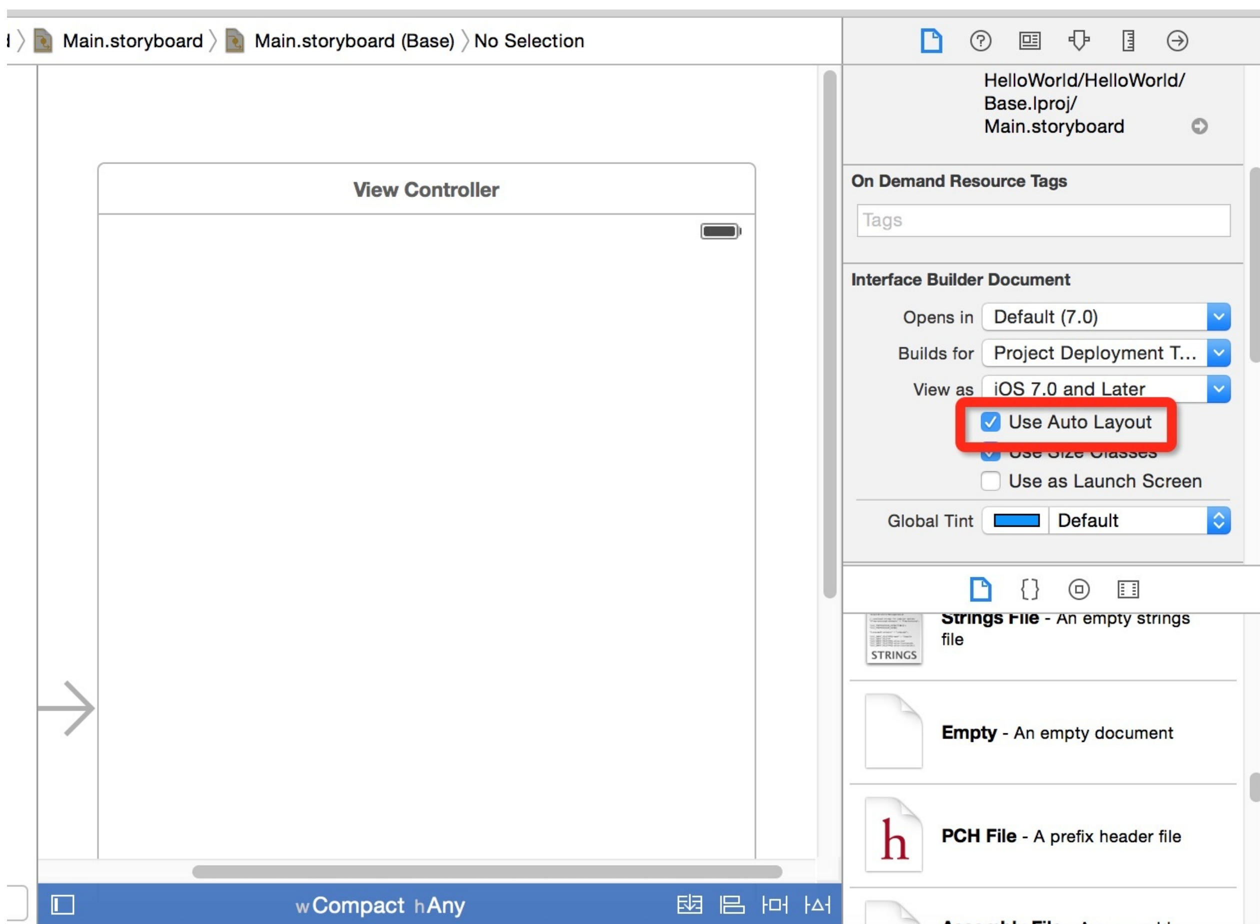
4、选择存放的目录，如果要使用Git源代码管理，将勾上Source Control的create git repository on My Mac. 点击create创建项目。



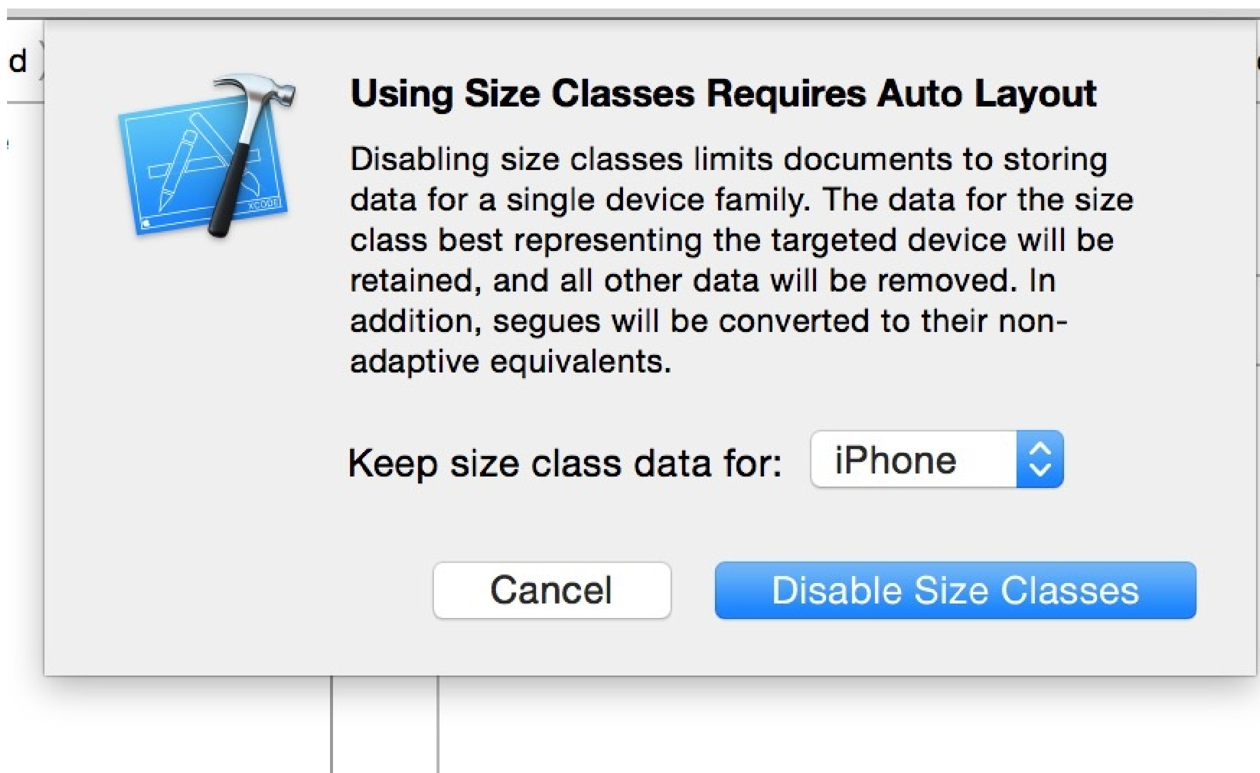
5、项目创建后，默认生成了一个示例文件，可以看到swift将oc中的h和m文件合并成了一个文件（即swift后缀名文件). Main.storyboard相当于xib文件，有比xib更多的功能。



6、打开main.storyboard,默认看到一个简单的空白的应用界面，大小为平板界面大小。 如果开发都只需要开发兼容iphone手机的app,那么可以把Use Auto Layout的勾去掉（默认为勾上）。



7、弹出了一个对话框，让我们选择界面尺寸，iPhone或都 iPad。我们选择iPhone的尺寸。

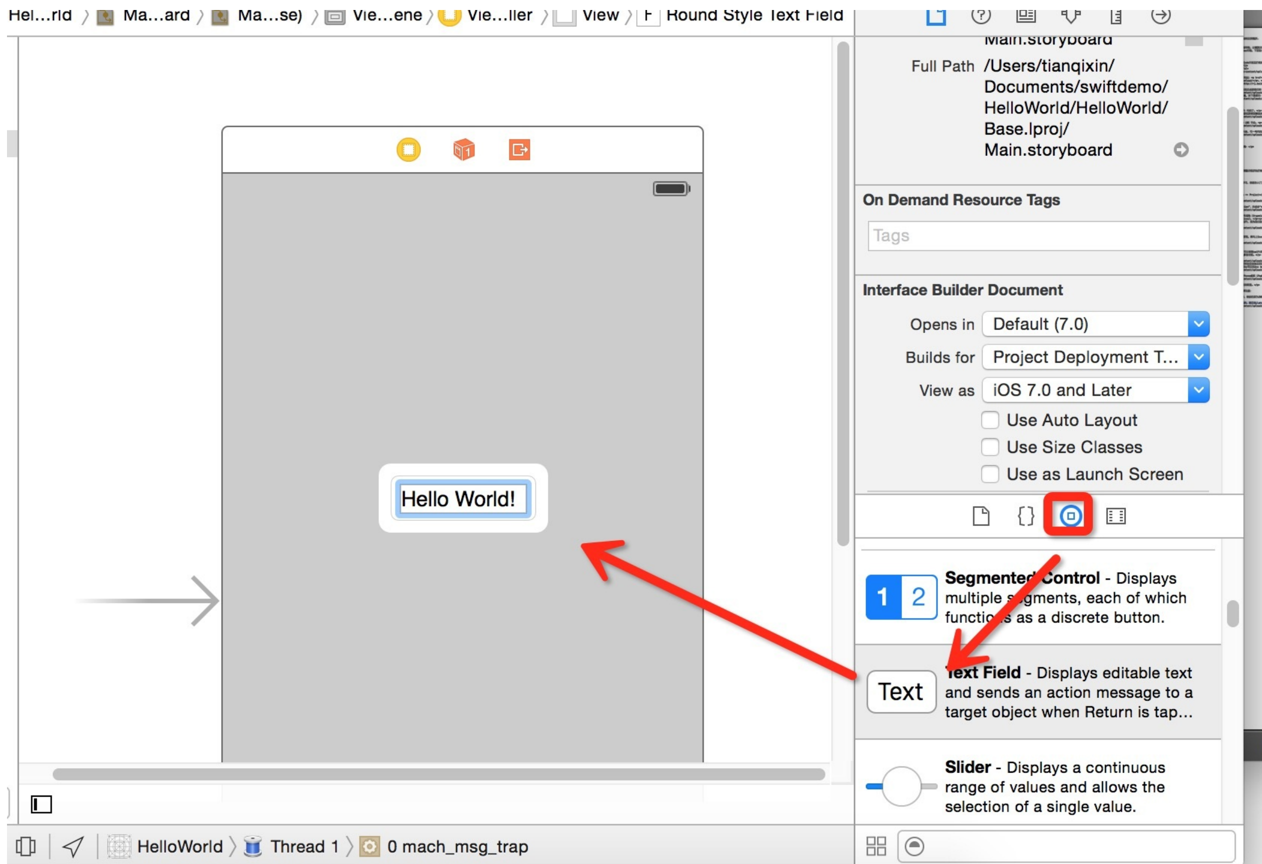


8、可以看到，界面大小变为了手机iphone的宽度和高度。

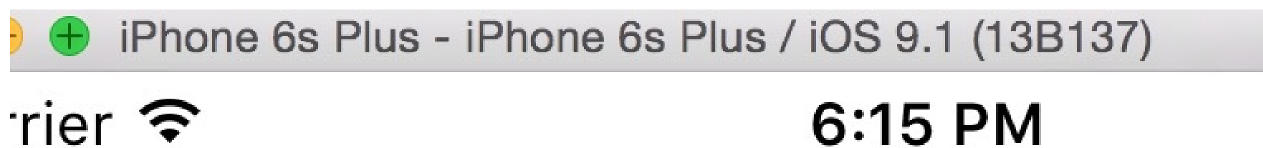
大家可以记住界面相关的尺寸，方便以后布局计算位置：

iPhone或iTouch的宽为320像素，高为480像素，状态栏高为20像素，toobar高为44像素，tabbar高为49像素，导航栏高为44像素。

9.我们为界面添加点内容，在右下方找到Text控件，将它拖入storyboard上，并双击写入文本"Hello World!"。



运行一下模拟器（command+R 快捷键或在菜单栏中选择 Product => Run)。



至此，我们的第一个Swift项目就完成了。

Swift 基本语法

在上一章节中我们已经讲到如何创建 Swift 语言的 "Hello, World!" 程序。现在我们来复习下。

如果创建的是 OS X playground 需要引入 Cocoa ：

```
import Cocoa

/* 我的第一个 Swift 程序 */
var myString = "Hello, World!"

print(myString)
```

如果我们想创建 iOS playground 则需要引入 UIKit :

```
import UIKit
var myString = "Hello, World!"
print(myString)
```

执行以上程序，输出结果为：

```
Hello, World!
```

以上代码即为 Swift 程序的基本结构，接下来我们来详细说明结构的组成部分。

Swift 引入

我们可以使用 **import** 语句来引入任何的 Objective-C 框架（或 C 库）到 Swift 程序中。例如 **import cocoa** 语句导入了使用了 Cocoa 库和API，我们可以在 Swift 程序中使用他们。

Cocoa 本身由 Objective-C 语言写成，Objective-C 又是 C 语言的严格超集，所以在 Swift 应用中我们可以很简单的混入 C 语言代码，甚至是 C++ 代码。

Swift 标记

Swift 程序由多种标记组成，标记可以是单词，标识符，常量，字符串或符号。例如以下 Swift 程序由三种标记组成：

```
print("test!")
标记是：单词、符号
print
(
   "test!"
)
```

注释

Swift的注释与C语言极其相似，单行注释以两个反斜线开头：

```
//这是一行注释
```

多行注释以/开始，以/结束:

```
/* 这也是一条注释,
但跨越多行 */
```

与 C 语言的多行注释有所不同的是，Swift 的多行注释可以嵌套在其他多行注释内部。写法是在一个多行注释块内插入另一个多行注释。第二个注释块封闭时，后面仍然接着第一个注释块：

```
/* 这是第一个多行注释的开头

/* 这是嵌套的第二个多行注释 */

这是第一个多行注释的结尾 */
```

多行注释的嵌套是你可以更快捷方便的注释代码块，即使代码块中已经有了注释。

分号

与其它语言不同的是，Swift不要求在每行语句的结尾使用分号(;)，但当你在同一行书写多条语句时，必须用分号隔开：

```
import Cocoa
/* 我的第一个 Swift 程序 */
var myString = "Hello, World!"; print(myString)
```

标识符

标识符就是给变量、常量、方法、函数、枚举、结构体、类、协议等指定的名字。构成标识符的字母均有一定的规范，Swift语言中标识符的命名规则如下：

- 区分大小写，Myname与myname是两个不同的标识符；
- 标识符首字符可以以下划线（_）或者字母开始，但不能是数字；
- 标识符中其他字符可以是下划线（_）、字母或数字。

例如：userName、User_Name、_sys_val、身高等为合法的标识符，而2mail、room#和class为非法的标识符。

注意:Swift中的字母采用的是Unicode编码[1]。Unicode叫做统一编码制，它包含了亚洲文字编码，如中文、日文、韩文等字符，甚至是我们在聊天工具中使用的表情符号

如果一定要使用关键字作为标识符，可以在关键字前后添加重音符号（`），例如：

关键字

关键字是类似于标识符的保留字符序列，除非用重音符号（`）将其括起来，否则不能用作标识符。关键字是对编译器具有特殊意义的预定义保留标识符。常见的关键字有以下4种。

与声明有关的关键字

class	deinit	enum	extension
func	import	init	internal
let	operator	private	protocol
public	static	struct	subscript
typealias	var		

与语句有关的关键字

break	case	continue	default
do	else	fallthrough	for
if	in	return	switch
where	while		

表达式和类型关键字

as	dynamicType	false	is
nil	self	Self	super
true	*COLUMN*	*FILE*	*FUNCTION*
LINE			

在特定上下文中使用的关键字

associativity	convenience	dynamic	didSet
final	get	infix	inout
lazy	left	mutating	none
nonmutating	optional	override	postfix
precedence	prefix	Protocol	required
right	set	Type	unowned
weak	willSet		

Swift 空格

Swift语言并不是像C/C++，Java那样完全忽视空格，Swift对空格的使用有一定的要求，但是又不像Python对缩进的要求那么严格。

在Swift中，运算符不能直接跟在变量或常量的后面。例如下面的代码会报错：

```
let a= 1 + 2
```

错误信息是：

```
error: prefix/postfix '=' is reserved
```

意思大概是等号直接跟在前面或后面这种用法是保留的。

下面的代码还是会报错（继续注意空格）：

```
let a = 1+ 2
```

错误信息是：

```
error: consecutive statements on a line must be separated by ';'
```

这是因为Swift认为到1+这个语句就结束了，2就是下一个语句了。

只有这样写才不会报错：

```
let a = 1 + 2;  // 编码规范推荐使用这种写法
let b = 3+4 // 这样也是OK的
```

Swift 字面量

所谓字面量，就是指像特定的数字，字符串或者是布尔值这样，能够直接了当地指出自己的类型并为变量进行赋值的值。比如在下面：

```
42                  // 整型字面量
3.14159             // 浮点型字面量
"Hello, world!"     // 字符串型字面量
true                // 布尔型字面量
```

Swift 数据类型

在我们使用任何程序语言编程时，需要使用各种数据类似来存储不同的信息。

变量的数据类型决定了如何将代表这些值的位存储到计算机的内存中。在声明变量时也可指定它的数据类型。

所有变量都具有数据类型，以决定能够存储哪种数据。

内置数据类型

Swift 提供了非常丰富的数据类型，以下列出了常用了集中数据类型：

Int

一般来说，你不需要专门指定整数的长度。Swift 提供了一个特殊的整数类型 `Int` ，长度与当前平台的原生字长相同：

- 在32位平台上， `Int` 和 `Int32` 长度相同。
- 在64位平台上， `Int` 和 `Int64` 长度相同。

除非你需要特定长度的整数，一般来说使用 `Int` 就够了。这可以提高代码一致性和可复用性。即使是在32位平台上， `Int` 可以存储的整数范围也可以达到 `-2,147,483,648` ~ `2,147,483,647` ，大多数时候这已经足够大了。

UInt

Swift 也提供了一个特殊的无符号类型 `UInt` ，长度与当前平台的原生字长相同：

- 在32位平台上， `UInt` 和 `UInt32` 长度相同。
- 在64位平台上， `UInt` 和 `UInt64` 长度相同。

> 注意：
>
> 尽量不要使用 `UInt` ，除非你真的需要存储一个和当前平台原生字长相同的无符号整数。除了这种情况，最好使用 `Int` ，即使你要存储的值已知是非负的。统一使用 `Int` 可以提高代码的可复用性，避免不同类型数字之间的转换，并且匹配数字的类型推断，请参考类型安全和类型推断。

浮点数

浮点数是有小数部分的数字，比如 `3.14159` ， `0.1` 和 `-273.15` 。

浮点类型比整数类型表示的范围更大，可以存储比 `Int` 类型更大或者更小的数字。Swift 提供了两种有符号浮点数类型：

- `Double` 表示64位浮点数。当你需要存储很大或者很高精度的浮点数时请使用此类型。
- `Float` 表示32位浮点数。精度要求不高的话可以使用此类型。

> 注意：
>
> `Double` 精确度很高，至少有15位数字，而 `Float` 最少只有6位数字。选择哪个类型取决于你的代码需要处理的值的范围。

布尔值

Swift 有一个基本的布尔（Boolean）类型，叫做Bool。布尔值指逻辑上的值，因为它们只能是真或者假。Swift 有两个布尔常量，true和false。

字符串

字符串是字符的序列集合，例如：

```
"Hello, World!"
```

字符

字符指的是单个字母，例如：

```
"C"
```

可选类型

使用可选类型（optionals）来处理值可能缺失的情况。可选类型表示有值或没有值。

数值范围

下表显示了不同变量类型内存的存储空间，及变量类型的最大最小值：

类型	大小（字节）	区间值
Int8	1 字节	-127 到 127
UInt8	1 字节	0 到 255
Int32	4 字节	-2147483648 到 2147483647
UInt32	4 字节	0 到 4294967295
Int64	8 字节	-9223372036854775808 到 9223372036854775807
UInt64	8 字节	0 到 18446744073709551615
Float	4 字节	1.2E-38 到 3.4E+38 (~6 digits)
Double	8 字节	2.3E-308 到 1.7E+308 (~15 digits)

类型别名

类型别名对当前的类型定义了另一个名字，类型别名通过使用 typealias 关键字来定义。语法格式如下：

```
typealias newname = type
```

例如以下定义了 Int 的类型别名为 Feet：

```
typealias Feet = Int
```

现在，我们可以通过别名来定义变量：

```
import Cocoa

typealias Feet = Int
var distance: Feet = 100
print(distance)
```

我们使用 playground 执行以上程序，输出结果为：

```
100
```

类型安全

Swift 是一个类型安全（type safe）的语言。

由于 Swift 是类型安全的，所以它会在编译你的代码时进行类型检查（type checks），并把不匹配的类型标记为错误。这可以让你在开发的时候尽早发现并修复错误。

```
import Cocoa

var varA = 42
varA = "This is hello"
print(varA)
```

以上程序，会在 Xcode 中报错：

```
error: cannot assign value of type 'String' to type 'Int'
varA = "This is hello"
```

意思为不能将 'String' 字符串赋值给 'Int' 变量。

类型推断

当你要处理不同类型的值时，类型检查可以帮你避免错误。然而，这并不是说你每次声明常量和变量的时候都需要显式指定类型。

如果你没有显式指定类型，Swift 会使用类型推断（type inference）来选择合适的类型。

例如，如果你给一个新常量赋值42并且没有标明类型，Swift 可以推断出常量类型是Int，因为你给它赋的初始值看起来像一个整数：

```
let meaningOfLife = 42
// meaningOfLife 会被推测为 Int 类型
```

同理，如果你没有给浮点字面量标明类型，Swift 会推断你想要的是Double：

```
let pi = 3.14159
// pi 会被推测为 Double 类型
```

当推断浮点数的类型时，Swift 总是会选择Double而不是Float。

如果表达式中同时出现了整数和浮点数，会被推断为Double类型：

```
let anotherPi = 3 + 0.14159
// anotherPi 会被推测为 Double 类型
```

原始值3没有显式声明类型，而表达式中出现了一个浮点字面量，所以表达式会被推断为Double类型。

实例

```
import Cocoa

// varA 会被推测为 Int 类型
var varA = 42
print(varA)

// varB 会被推测为 Double 类型
var varB = 3.14159
print(varB)

// varC 也会被推测为 Double 类型
var varC = 3 + 0.14159
print(varC)
```

执行以上代码，输出结果为：

```
42
3.14159
3.14159
```

Swift 变量

变量是一种使用方便的占位符，用于引用计算机内存地址。

Swift 每个变量都指定了特定的类型，该类型决定了变量占用内存的大小，不同的数据类型也决定可存储值的范围。

上一章节我们已经为大家介绍了基本的数据类型，包括整形Int、浮点数Double和Float、布尔类型Bool以及字符串类型String。此外，Swift还提供了其他更强大数据类型， Optional, Array, Dictionary, Struct, 和 Class 等。

接下来我们将为大家介绍如何在 Swift 程序中声明和使用变量。

变量声明

变量声明意思是告诉编译器在内存中的哪个位置上为变量创建多大的存储空间。

在使用变量前，你需要使用 **var** 关键字声明它，如下所示：

```
var variableName = <initial value>
```

以下是一个 Swift 程序中变量声明的简单实例：

```
import Cocoa

var varA = 42
print(varA)

var varB:Float

varB = 3.14159
print(varB)
```

以上程序执行结果为：

```
42
3.14159
```

变量命名

变量名可以由字母，数字和下划线组成。

变量名需要以字母或下划线开始。

Swift 是一个区分大小写的语言，所以字母大写与小写是不一样的。

变量名也可以使用简单的 Unicode 字符，如下实例：

```
import Cocoa

var _var = "Hello, Swift!"
print(_var)

var 你好 = "你好世界"
var 菜鸟教程 = "www.runoob.com"
print(你好)
print(菜鸟教程)
```

以上程序执行结果为：

```
Hello, Swift!
你好世界
www.runoob.com
```

变量输出

变量和常量可以使用 **print**（swift 2 将 print 替换了 println） 函数来输出。

在字符串中可以使用括号与反斜线来插入变量，如下实例：

```
import Cocoa

var name = "菜鸟教程"
var site = "http://www.runoob.com"

print("\(name)的官网地址为：\(site)")
```

以上程序执行结果为：

```
菜鸟教程的官网地址为：http://www.runoob.com
```

Swift 可选(Optionals)类型

Swift 的可选（Optional）类型，用于处理值缺失的情况。可选表示"那儿有一个值，并且它等于 x "或者"那儿没有值"。

Swfit语言定义后缀？作为命名类型Optional的简写，换句话说，以下两种声明是相等的：

```
var optionalInteger: Int?
var optionalInteger: Optional<Int>
```

在这两种情况下，变量optionalInteger都是可选整数类型。注意，在类型和？之间没有空格。

Optional 是一个含有两种情况的枚举，None和Some(T)，用来表示可能有或可能没有值。任何类型都可以明确声明为（或者隐式转换）可选类型。当声明一个可选类型的时候，要确保用括号给？操作符一个合适的范围。例如，声明可选整数数组，应该写成(Int[])?；写成Int[]?会报错。

当你声明一个可选变量或者可选属性的时候没有提供初始值，它的值会默认为nil。

可选项遵照LogicValue协议，因此可以出现在布尔环境中。在这种情况下，如果可选类型T?包含类型为T的任何值（也就是说它的值是Optional.Some(T)），这个可选类型等于true，反之为false。

如果一个可选类型的实例包含一个值，你可以用后缀操作符 ！来访问这个值，如下所示：

```
optionalInteger = 42
optionalInteger! // 42
```

使用操作符！去获取值为nil的可选变量会有运行时错误。

你可以用可选链接和可选绑定选择性执行可选表达式上的操作。如果值为nil，任何操作都不会执行，也不会有运行报错。

让我们来详细看下以下实例来了解 Swift 中可选类型的应用:

```
import Cocoa

var myString:String? = nil

if myString != nil {
    print(myString)
}else{
    print("字符串为 nil")
}
```

以上程序执行结果为：

```
字符串为 nil
```

可选类型类似于Objective-C中指针的nil值，但是nil只对类(class)有用，而可选类型对所有的类型都可用，并且更安全。

强制解析

当你确定可选类型确实包含值之后，你可以在可选的名字后面加一个感叹号（!）来获取值。这个感叹号表示"我知道这个可选有值，请使用它。"这被称为可选值的强制解析（forced unwrapping）。

实例如下：

```
import Cocoa

var myString:String?

myString = "Hello, Swift!"

if myString != nil {
   print(myString)
}else{
   print("myString 值为 nil")
}
```

以上程序执行结果为：

```
Optional("Hello, Swift!")
```

强制解析可选值，使用感叹号（!）：

```
import Cocoa

var myString:String?

myString = "Hello, Swift!"

if myString != nil {
   // 强制解析
   print( myString! )
}else{
   print("myString 值为 nil")
}
```

以上程序执行结果为：

```
Hello, Swift!
```

> 注意：
>
> 使用 `!` 来获取一个不存在的可选值会导致运行时错误。使用 `!` 来强制解析值之前，一定要确定可选包含一个非 `nil` 的值。

自动解析

你可以在声明可选变量时使用感叹号（!）替换问号（?）。这样可选变量在使用时就不需要再加一个感叹号（!）来获取值，它会自动解析。

实例如下：

```
import Cocoa

var myString:String!

myString = "Hello, Swift!"

if myString != nil {
   print(myString)
}else{
   print("myString 值为 nil")
}
```

以上程序执行结果为：

```
Hello, Swift!
```

可选绑定

使用可选绑定（optional binding）来判断可选类型是否包含值，如果包含就把值赋给一个临时常量或者变量。可选绑定可以用在if和while语句中来对可选类型的值进行判断并把值赋给一个常量或者变量。

像下面这样在if语句中写一个可选绑定：

```
if let constantName = someOptional {
    statements
}
```

让我们来看下一个简单的可选绑定实例：

```
import Cocoa

var myString:String?

myString = "Hello, Swift!"

if let yourString = myString {
   print("你的字符串值为 - \(yourString)")
}else{
   print("你的字符串没有值")
}
```

以上程序执行结果为：

```
你的字符串值为 - Hello, Swift!
```

Swift 常量

常量一旦设定，在程序运行时就无法改变其值。

常量可以是任何的数据类型如：整型常量，浮点型常量，字符常量或字符串常量。同样也有枚举类型的常量：

常量类似于变量，区别在于常量的值一旦设定就不能改变，而变量的值可以随意更改。

常量声明

常量使用关键字 **let** 来声明，语法如下：

```
let constantName = <initial value>
```

以下是一个简单的 Swift 程序中使用常量的实例：

```
import Cocoa

let constA = 42
print(constA)
```

以上程序执行结果为：

```
42
```

类型标注

当你声明常量或者变量的时候可以加上类型标注（type annotation），说明常量或者变量中要存储的值的类型。如果要添加类型标注，需要在常量或者变量名后面加上一个冒号和空格，然后加上类型名称。

```
var constantName:<data type> = <optional initial value>
```

以下是一个简单是实例演示了 Swift 中常量使用类型标注。需要注意的是常量定义时必须初始值：

```
import Cocoa

let constA = 42
print(constA)

let constB:Float = 3.14159

print(constB)
```

以上程序执行结果为：

```
42
3.14159
```

常量命名

常量的命名可以由字母，数字和下划线组成。

常量需要以字母或下划线开始。

Swift 是一个区分大小写的语言，所以字母大写与小写是不一样的。

常量名也可以使用简单的 Unicode 字符，如下实例：

```
import Cocoa

let _const = "Hello, Swift!"
print(_const)

let 你好 = "你好世界"
print(你好)
```

以上程序执行结果为：

```
Hello, Swift!
你好世界
```

常量输出

变量和常量可以使用 print（swift 2 将 print 替换了 println） 函数来输出。

在字符串中可以使用括号与反斜线来插入常量，如下实例：

```
import Cocoa

let name = "菜鸟教程"
let site = "http://www.runoob.com"

print("\(name)的官网地址为：\(site)")
```

以上程序执行结果为：

```
菜鸟教程的官网地址为：http://www.runoob.com
```

Swift 字面量

所谓字面量，就是指像特定的数字，字符串或者是布尔值这样，能够直接了当地指出自己的类型并为变量进行赋值的值。比如在下面：

```
let aNumber = 3          //整型字面量
let aString = "Hello"    //字符串字面量
let aBool = true         //布尔值字面量
```

整型字面量

整型字面量可以是一个十进制，二进制，八进制或十六进制常量。 二进制前缀为 0b，八进制前缀为 0o，十六进制前缀为 0x，十进制没有前缀：

以下为一些整型字面量的实例：

```
let decimalInteger = 17           // 17 - 十进制表示
let binaryInteger = 0b10001       // 17 - 二进制表示
let octalInteger = 0o21           // 17 - 八进制表示
let hexadecimalInteger = 0x11     // 17 - 十六进制表示
```

浮点型字面量

浮点型字面量有整数部分，小数点，小数部分及指数部分。

除非特别指定，浮点型字面量的默认推导类型为 Swift 标准库类型中的 Double，表示64位浮点数。

浮点型字面量默认用十进制表示（无前缀），也可以用十六进制表示（加前缀 0x）。

> 十进制浮点型字面量由十进制数字串后跟小数部分或指数部分（或两者皆有）组成。十进制小数部分由小数点 . 后跟十进制数字串组成。指数部分由大写或小写字母 e 为前缀后跟十进制数字串组成，这串数字表示 e 之前的数量乘以 10 的几次方。例如：1.25e2 表示 1.25 × 10^2，也就是 125.0；同样，1.25e－2 表示 1.25 × 10^－2，也就是 0.0125。
>
> 十六进制浮点型字面量由前缀 0x 后跟可选的十六进制小数部分以及十六进制指数部分组成。十六进制小数部分由小数点后跟十六进制数字串组成。指数部分由大写或小写字母 p 为前缀后跟十进制数字串组成，这串数字表示 p 之前的数量乘以 2 的几次方。例如：0xFp2 表示 15 × 2^2，也就是 60；同样，0xFp-2 表示 15 × 2^-2，也就是 3.75。
>
> 负的浮点型字面量由一元运算符减号 - 和浮点型字面量组成，例如 -42.5。

浮点型字面量允许使用下划线 _ 来增强数字的可读性，下划线会被系统忽略，因此不会影响字面量的值。同样地，也可以在数字前加 0，并不会影响字面量的值。

以下为一些浮点型字面量的实例：

```
let decimalDouble = 12.1875       //十进制浮点型字面量
let exponentDouble = 1.21875e1    //十进制浮点型字面量
let hexadecimalDouble = 0xC.3p0   //十六进制浮点型字面量
```

字符串型字面量

字符串型字面量由被包在双引号中的一串字符组成，形式如下：

```
"characters"
```

字符串型字面量中不能包含未转义的双引号 （"）、未转义的反斜线（\）、回车符或换行符。

转移字符	含义
\0	空字符
\\	反斜线 \
\b	退格(BS) ，将当前位置移到前一列
\f	换页(FF)，将当前位置移到下页开头
\n	换行符
\r	回车符
\t	水平制表符
\v	垂直制表符
\'	单引号
\"	双引号
\000	1到3位八进制数所代表的任意字符
\xhh...	1到2位十六进制所代表的任意字符

以下为字符串字面量的简单实例：

```
import Cocoa

let stringL = "Hello\tWorld\n\n菜鸟教程官网：\'http://www.runoob.com\
print(stringL)
```

以上程序执行结果为：

```
Hello	World

菜鸟教程官网：'http://www.runoob.com'
```

布尔型字面量

布尔型字面量的默认类型是 Bool。

布尔值字面量有三个值，它们是 Swift 的保留关键字：

- **true** 表示真。
- **false** 表示假。
- **nil** 表示没有值。

Swift 运算符

运算符是一个符号，用于告诉编译器执行一个数学或逻辑运算。

Swift 提供了以下几种运算符：

- 算术运算符
- 比较运算符
- 逻辑运算符
- 位运算符
- 赋值运算符
- 区间运算符
- 其他运算符

本章节我们将为大家详细介绍算术运算符、关系运算符、逻辑运算符、位运算符、赋值运算符及其他运算符。

算术运算符

以下表格列出了 Swift 语言支持的算术运算符，其中变量 A 为 10，变量 B 为 20：

运算符	描述	实例
+	加号	A + B 结果为 30
−	减号	A − B 结果为 -10
*	乘号	A * B 结果为 200
/	除号	B / A 结果为 2
%	求余	B % A 结果为 0
++	自增	A++ 结果为 11
--	自减	A-- 结果为 9

实例

以下为算术运算的简单实例：

```
import Cocoa

var A = 10
var B = 20

print("A + B 结果为：\(A + B)")
print("A - B 结果为：\(A - B)")
print("A * B 结果为：\(A * B)")
print("B / A 结果为：\(B / A)")
A++
print("A++ 后 A 的值为 \(A)")
B--
print("B-- 后 B 的值为 \(B)")
```

以上程序执行结果为：

```
A + B 结果为：30
A - B 结果为：-10
A * B 结果为：200
B / A 结果为：2
A++ 后 A 的值为 11
B-- 后 B 的值为 19
```

比较运算符

以下表格列出了 Swift 语言支持的比较运算符，其中变量 A 为 10，变量 B 为 20：

运算符	描述	实例
==	等于	(A == B) 为 false。
!=	不等于	(A != B) 为 true。
>	大于	(A > B) 为 false。
<	小于	(A < B) 为 true。
>=	大于等于	(A >= B) 为 false。
<=	小于等于	(A <= B) 为 true。

实例

以下为比较运算的简单实例：

```
import Cocoa

var A = 10
var B = 20

print("A == B 结果为：\(A == B)")
print("A != B 结果为：\(A != B)")
print("A > B 结果为：\(A > B)")
print("A < B 结果为：\(A < B)")
print("A >= B 结果为：\(A >= B)")
print("A <= B 结果为：\(A <= B)")
```

以上程序执行结果为：

```
A == B 结果为：false
A != B 结果为：true
A > B 结果为：false
A < B 结果为：true
A >= B 结果为：false
A <= B 结果为：true
```

逻辑运算符

以下表格列出了 Swift 语言支持的逻辑运算符，其中变量 A 为 true，变量 B 为 false：

运算符	描述	实例
&&	逻辑与。如果运算符两侧都为 TRUE 则为 TRUE。	(A && B) 为 false。
||	逻辑或。 如果运算符两侧至少有一个为 TRUE 则为 TRUE。	(A || B) 为 true。
!	逻辑非。布尔值取反，使得true变false，false变true。	!(A && B) 为 true。

以下为逻辑运算的简单实例：

```
import Cocoa

var A = true
var B = false

print("A && B 结果为：\(A && B)")
print("A || B 结果为：\(A || B)")
print("!A 结果为：\(!A)")
print("!B 结果为：\(!B)")
```

以上程序执行结果为：

```
A && B 结果为：false
A || B 结果为：true
!A 结果为：false
!B 结果为：true
```

位运算符

位运算符用来对二进制位进行操作，~,&,|,^分别为取反，按位与与，按位与或，按位与异或运算，如下表实例：

p	q	p & q	p | q	p ^ q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

如果指定 A = 60; 及 B = 13; 两个变量对应的二进制为：

```
A = 0011 1100

B = 0000 1101
```

进行位运算：

运算符	描述	图解	实例
	按位与。按位与运算符对两个数		

&	进行操作，然后返回一个新的数，这个数的每个位都需要两个输入数的同一位都为1时才为1。		(A & B) 结果为 12, 二进制为 0000 1100
|	按位或。按位或运算符|比较两个数，然后返回一个新的数，这个数的每一位设置1的条件是两个输入数的同一位都不为0(即任意一个为1，或都为1)。		(A | B) 结果为 61, 二进制为 0011 1101
^	按位异或. 按位异或运算符^比较两个数，然后返回一个数，这个数的每个位设为1的条件是两个输入数的同一位不同，如果相同就设为0。		(A ^ B) 结果为 49, 二进制为 0011 0001
~	按位取反运算符~对一个操作数的每一位都取反。		(~A) 结果为 -61, 二进制为 1100 0011 in 2's complement form.
<<	按位左移。左移操作符（<<）将操作数的所有位向左移动指定的位数。	下图展示了11111111 << 1（11111111 左移一位）的结果。蓝色数字表示被移动位，灰色表示被丢弃位，空位用橙色的0填充。	A << 2 结果为 240, 二进制为 1111 0000
	按位右移。右移	下图展示了11111111 >> 1（11111111 右移一位）的结果。蓝色数字表示被移动位，灰色表示被丢弃位，空位用橙色的0填	

>>	操作符（<<）将操作数的所有位向又移动指定的位数。	充。	为 15, 二进制为 0000 1111

以下为位运算的简单实例：

```
import Cocoa

var A = 60  // 二进制为 0011 1100
var B = 13 // 二进制为 0000 1101

print("A&B 结果为：\(A&B)")
print("A|B 结果为：\(A|B)")
print("A^B 结果为：\(A^B)")
print("~A 结果为：\(~A)")
```

以上程序执行结果为：

```
A&B 结果为：12
A|B 结果为：61
A^B 结果为：49
~A 结果为：-61
```

赋值运算

下表列出了 Swift 语言的基本赋值运算：

运算符	描述	实例
=	简单的赋值运算，指定右边操作数赋值给左边的操作数。	C = A + B 将 A + B 的运算结果赋值给 C
+=	相加后再赋值，将左右两边的操作数相加后再赋值给左边的操作数。	C += A 相当于 C = C + A
-=	相减后再赋值，将左右两边的操作数相减后再赋值给左边的操作数。	C -= A 相当于 C = C - A
*=	相乘后再赋值，将左右两边的操作数相乘后再赋值给左边的操作数。	C = A 相当于 C = C A
/=	相除后再赋值，将左右两边的操作数相除后再赋值给左边的操作数。	C /= A 相当于 C = C / A
%=	求余后再赋值，将左右两边的操作数求余后再赋值给左边的操作数。	C %= A is equivalent to C = C % A
<<=	按位左移后再赋值	C <<= 2 相当于 C = C << 2
>>=	按位右移后再赋值	C >>= 2 相当于 C = C >> 2
&=	按位与运算后赋值	C &= 2 相当于 C = C & 2
^=	按位异或运算符后再赋值	C ^= 2 相当于 C = C ^ 2
|=	按位或运算后再赋值	C |= 2 相当于 C = C | 2

以下为赋值运算的简单实例：

```
import Cocoa

var A = 10
var B = 20
var C = 100

C = A + B
print("C 结果为：\(C)")

C += A
print("C 结果为：\(C)")

C -= A
print("C 结果为：\(C)")

C *= A
print("C 结果为：\(C)")

C /= A
print("C 结果为：\(C)")

//以下测试已注释，可去掉注释测试每个实例
/*
C %= A
print("C 结果为：\(C)")

C <<= A
print("C 结果为：\(C)")

C >>= A
print("C 结果为：\(C)")

C &= A
print("C 结果为：\(C)")

C ^= A
print("C 结果为：\(C)")

C |= A
print("C 结果为：\(C)")
*/
```

以上程序执行结果为：

```
C 结果为：30
C 结果为：40
C 结果为：30
C 结果为：300
C 结果为：30
```

区间运算符

Swift 提供了两个区间的运算符。

运算符	描述	实例
闭区间运算符	闭区间运算符（a...b）定义一个包含从a到b(包括a和b)的所有值的区间，b必须大于等于a。 闭区间运算符在迭代一个区间的所有值时是非常有用的，如在for-in循环中：	1...5 区间值为 1, 2, 3, 4 和 5
半开区间运算符	半开区间（a..<b）定义一个从a到b但不包括b的区间。 之所以称为半开区间，是因为该区间包含第一个值而不包括最后的值。<="" td=""></b）定义一个从a到b但不包括b的区间。>	1..< 5 区间值为 1, 2, 3, 和 4

以下为区间运算的简单实例：

```
import Cocoa

print("闭区间运算符:")
for index in 1...5 {
    print("\(index) * 5 = \(index * 5)")
}

print("半开区间运算符:")
for index in 1..<5 {
    print("\(index) * 5 = \(index * 5)")
}
```

以上程序执行结果为：

```
闭区间运算符:
1 * 5 = 5
2 * 5 = 10
3 * 5 = 15
4 * 5 = 20
5 * 5 = 25
半开区间运算符:
1 * 5 = 5
2 * 5 = 10
3 * 5 = 15
4 * 5 = 20
```

其他运算符

Swift 提供了其他类型的的运算符，如一元、二元和三元运算符。

- 一元运算符对单一操作对象操作（如 `-a` ）。一元运算符分前置运算符和后置运算符，前置运算符需紧跟在操作对象之前（如 `!b` ），后置运算符需紧跟在操作对象之后（如 `i++` ）。
- 二元运算符操作两个操作对象（如 `2 + 3` ），是中置的，因为它们出现在两个操作对象之间。
- 三元运算符操作三个操作对象，和 C 语言一样，Swift 只有一个三元运算符，就是三目运算符（ `a ? b : c` ）。

运算符	描述	实例
一元减	数字前添加 - 号前缀	-3 或 -4
一元加	数字钱添加 + 号前缀	+6 结果为 6
三元运算符	条件 ? X : Y	如果添加为 true，值为 X，否则为 Y

以下为一元、二元、三元的运算的简单实例：

```
import Cocoa

var A = 1
var B = 2
var C = true
var D = false
print("-A 的值为：\(-A)")
print("A + B 的值为：\(A + B)")
print("三元运算：\(C ? A : B )")
print("三元运算：\(D ? A : B )")
```

以上程序执行结果为：

```
-A 的值为：-1
A + B 的值为：3
三元运算：1
三元运算：2
```

运算符优先级

在一个表达式中可能包含多个有不同运算符连接起来的、具有不同数据类型的数据对象；由于表达式有多种运算，不同的运算顺序可能得出不同结果甚至出现错误运算错误，因为当表达式中含多种运算时，必须按一定顺序进行结合，才能保证运算的合理性和结果的正确性、唯一性。

优先级从上到下依次递减，最上面具有最高的优先级，逗号操作符具有最低的优先级。

相同优先级中，按结合顺序计算。大多数运算是从左至右计算，只有三个优先级是从右至左结合的，它们是单目运算符、条件运算符、赋值运算符。

基本的优先级需要记住：

- 指针最优，单目运算优于双目运算。如正负号。
- 先乘除（模），后加减。
- 先算术运算，后移位运算，最后位运算。请特别注意：1 << 3 + 2 & 7 等价于 (1 << (3 + 2))&7
- 逻辑运算最后计算

运算符类型	运算符	结合方向
表达式运算	() [] . expr++ expr--	左到右
一元运算符	& + - ! ~ ++expr --expr * / % + - >> << < > <= >= == !=	右到左
位运算符	& ^ | && ||	左到右
三元运算符	?:	右到左
赋值运算符	= += -= *= /= %= >>= <<= &= ^= |=	右到左
逗号	,	左到右

以下为运算符优先级简单实例：

```
import Cocoa

var A = 0

A = 2 + 3 * 4 % 5
print("A 的值为：\(A)")
```

以上程序执行结果为：

```
A 的值为：4
```

实例解析：

根据运算符优先级，可以将以上程序的运算解析为以下步骤，表达式相当于：

```
2 + ((3 * 4) % 5)
```

第一步计算：(3 * 4) = 12，所以表达式相当于：

```
2 + (12 % 5)
```

第二步计算 12 % 5 = 2，所以表达式相当于：

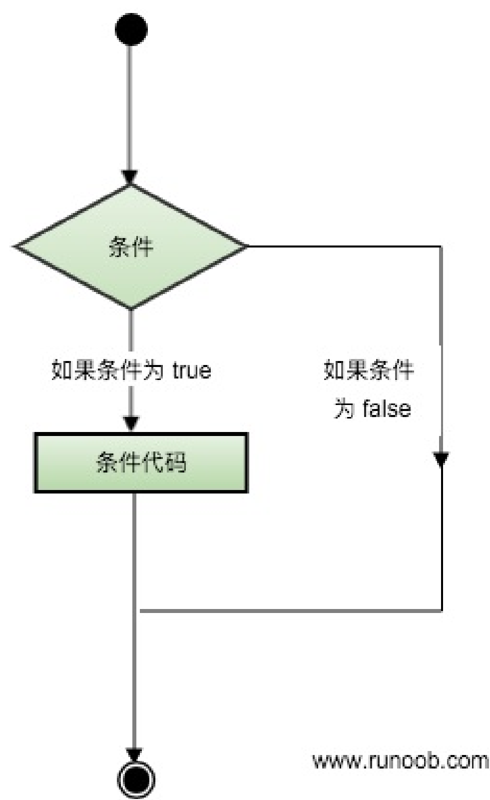
```
2 + 2
```

此时可以容易地看出计算的结果为 4。

Swift 条件语句

条件语句通过设定的一个或多个条件来执行程序，在条件为真时执行指定的语句，在条件为 false 时执行另外指定的语句。

可以通过下图来简单了解条件语句的执行过程:



Swift 提供了以下几种类型的条件语句：

语句	描述
if 语句	**if** 语句 由一个布尔表达式和一个或多个执行语句组成。
if...else 语句	**if** 语句 后可以有可选的 **else** 语句, **else** 语句在布尔表达式为 false 时执行。
if...else if...else 语句	**if** 后可以有可选的 **else if...else** 语句, **else if...else** 语句常用于多个条件判断。
内嵌 if 语句	你可以在 **if** 或 **else if** 中内嵌 **if** 或 **else if** 语句。
switch 语句	switch 语句允许测试一个变量等于多个值时的情况。

? : 运算符

我们已经在前面的章节中讲解了 条件运算符 **? :**，可以用来替代 **if...else** 语句。它的一般形式如下：

```
Exp1 ? Exp2 : Exp3;
```

其中，Exp1、Exp2 和 Exp3 是表达式。请注意，冒号的使用和位置。

? 表达式的值是由 Exp1 决定的。如果 Exp1 为真，则计算 Exp2 的值，结果即为整个 ? 表达式的值。如果 Exp1 为假，则计算 Exp3 的值，结果即为整个 ? 表达式的值。

Swift if 语句

一个 **if** 语句 由一个布尔表达式后跟一个或多个语句组成。

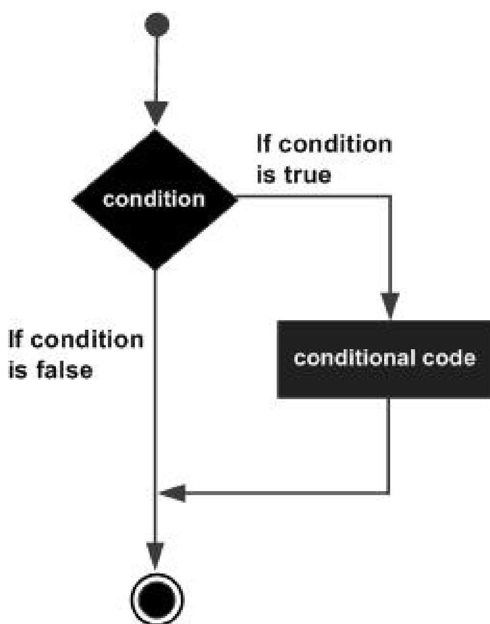
语法

Swift 语言中 **if** 语句的语法：

```
if boolean_expression {
   /* 如果布尔表达式为真将执行的语句 */
}
```

如果布尔表达式为 **true**，则 if 语句内的代码块将被执行。如果布尔表达式为 **false**，则 if 语句结束后的第一组代码（闭括号后）将被执行。

流程图



实例

```
import Cocoa

var varA:Int = 10;

/* 检测条件 */
if varA < 20 {
    /* 如果条件语句为 true 执行以下程序 */
    print("varA 小于 20");
}
print("varA 变量的值为 \(varA)");
```

当上面的代码被编译和执行时，它会产生下列结果：

```
varA 小于  20
varA 变量的值为  10
```

Swift if...else 语句

一个 **if** 语句 后可跟一个可选的 **else** 语句，else 语句在布尔表达式为 false 时执行。

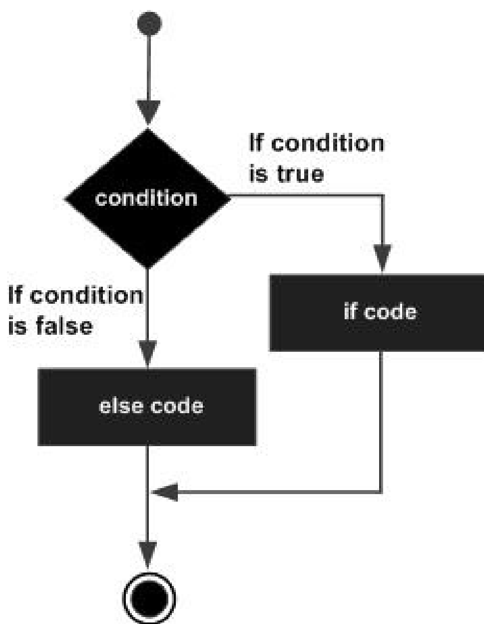
语法

Swift 语言中 **if...else** 语句的语法：

```
if boolean_expression {
   /* 如果布尔表达式为真将执行的语句 */
} else {
   /* 如果布尔表达式为假将执行的语句 */
}
```

如果布尔表达式为 **true**，则执行 **if** 块内的代码。如果布尔表达式为 **false**，则执行 **else** 块内的代码。

流程图



实例

```
import Cocoa

var varA:Int = 100;

/* 检测布尔条件 */
if varA < 20 {
   /* 如果条件为 true 执行以下语句 */
   print("varA 小于 20");
} else {
   /* 如果条件为 false 执行以下语句 */
   print("varA 大于 20");
}
print("varA 变量的值为 \(varA)");
```

当上面的代码被编译执行时，它会产生下列结果：

```
varA 大于 20
varA 变量的值为 100
```

Swift if...else if...else 语句

一个 **if** 语句 后可跟一个可选的 **else if...else** 语句，**else if...else** 语句 在测试多个条件语句时是非常有用的。

当你使用 if , else if , else 语句时需要注意以下几点：

- if 语句后可以有 0 个或 1 个 else，但是如果 有 else if 语句，else 语句需要在 else if 语句之后。
- if 语句后可以有 0 个或多个 else if 语句，else if 语句必须在 else 语句出现之前。
- 一旦 else 语句执行成功，其他的 else if 或 else 语句都不会执行。

语法

```
if boolean_expression_1 {
   /* 如果 boolean_expression_1 表达式为 true 则执行该语句 */
} else if boolean_expression_2 {
   /* 如果 boolean_expression_2 表达式为 true 则执行该语句 */
} else if boolean_expression_3 {
   /* 如果 boolean_expression_3 表达式为 true 则执行该语句 */
} else {
   /* 如果以上所有条件表达式都不为 true 则执行该语句 */
}
```

实例

```
import Cocoa

var varA:Int = 100;

/* 检测布尔条件 */
if varA == 20 {
    /* 如果条件为 true 执行以下语句 */
    print("varA 的值为 20");
} else if varA == 50 {
    /* 如果条件为 true 执行以下语句 */
    print("varA 的值为 50");
} else {
    /* 如果以上条件都为 false 执行以下语句 */
    print("没有匹配条件");
}
print("varA 变量的值为 \(varA)");
```

当上面的代码被编译执行时，它会产生下列结果：

```
没有匹配条件
varA 变量的值为  100
```

Swift 嵌套 if 语句

在 Swift 语言中，你可以在一个 if 或 else if 语句内使用另一个 if 或 else if 语句。

语法

Swift 语言中 嵌套 if 语句的语法：

```
import Cocoa

var varA:Int = 100;

/* 检测布尔条件 */
if varA == 20 {
    /* 如果条件为 true 执行以下语句 */
    print("varA 的值为 20");
} else if varA == 50 {
    /* 如果条件为 true 执行以下语句 */
    print("varA 的值为 50");
} else {
    /* 如果以上条件都为 false 执行以下语句 */
    print("没有匹配条件");
}
print("varA 变量的值为 \(varA)");
```

您可以以嵌套 **else if...else**，方式与嵌套 *if* 语句相似。

实例

```
import Cocoa

var varA:Int = 100;
var varB:Int = 200;

/* 检测布尔条件 */
if varA == 100 {
   /* 如果条件为 true 执行以下语句 */
   print("第一个条件为 true");

   if varB == 200 {
      /* 如果条件为 true 执行以下语句 */
      print("第二个条件也是 true");
   }
}
print("varA 变量的值为 \(varA)");
print("varB 变量的值为 \(varB)");
```

当上面的代码被编译执行时，它会产生下列结果：

```
第一个条件为 true
第二个条件也是 true
varA 变量的值为 100
varB 变量的值为 200
```

Swift switch 语句

switch 语句允许测试一个变量等于多个值时的情况。 Swift 语言中只要匹配到 case 语句，则整个 switch 语句执行完成。

语法

Swift 语言中 switch 语句的语法：

```
switch expression {
   case expression1  :
      statement(s)
      fallthrough /* 可选 */
   case expression2, expression3  :
      statement(s)
      fallthrough /* 可选 */

   default : /* 可选 */
      statement(s);
}
```

一般在 switch 语句中不使用 fallthrough 语句。

这里我们需要注意 case 语句中如果没有使用 **fallthrough** 语句，则在执行当前的 case 语句后，switch 会终止，控制流将跳转到 switch 语句后的下一行。

如果使用了**fallthrough** 语句，则会继续执行之后的 case 或 default 语句，不论条件是否满足都会执行。

> 注意：在大多数语言中，switch 语句块中，case 要紧跟 break，否则 case 之后的语句会顺序运行，而在 Swift 语言中，默认是不会执行下去的，switch 也会终止。如果你想在 Swift 中让 case 之后的语句会按顺序继续运行，则需要使用 fallthrough 语句。

实例1

以下实例没有使用 fallthrough 语句：

```
import Cocoa

var index = 10

switch index {
   case 100  :
      print( "index 的值为 100")
   case 10,15  :
      print( "index 的值为 10 或 15")
   case 5  :
      print( "index 的值为 5")
   default :
      print( "默认 case")
}
```

当上面的代码被编译执行时，它会产生下列结果：

```
index 的值为  10  或  15
```

实例2

以下实例使用 fallthrough 语句：

```
import Cocoa

var index = 10

switch index {
   case 100  :
      print( "index 的值为 100")
      fallthrough
   case 10,15  :
      print( "index 的值为 10 或 15")
      fallthrough
   case 5  :
      print( "index 的值为 5")
   default :
      print( "默认 case")
}
```

当上面的代码被编译执行时，它会产生下列结果：

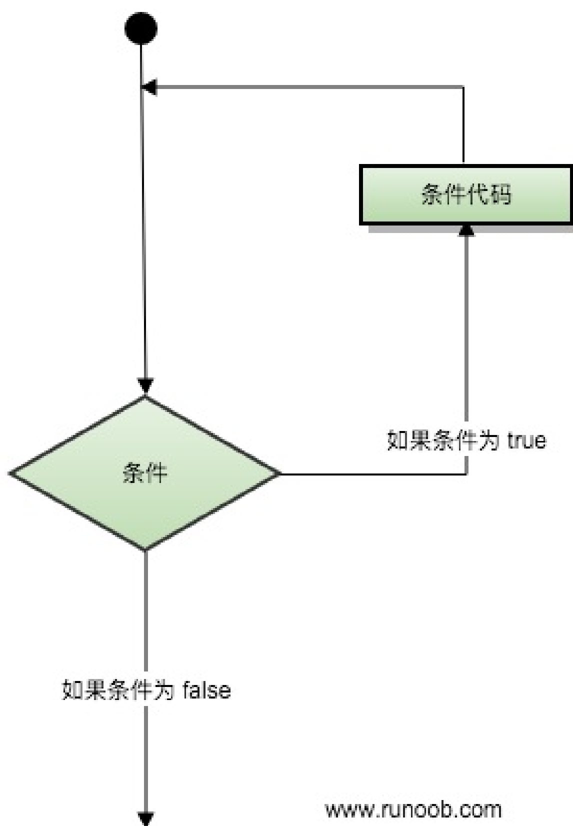
```
index 的值为  10  或  15
index 的值为  5
```

Swift 循环

有的时候，我们可能需要多次执行同一块代码。一般情况下，语句是按顺序执行的：函数中的第一个语句先执行，接着是第二个语句，依此类推。

编程语言提供了更为复杂执行路径的多种控制结构。

循环语句允许我们多次执行一个语句或语句组，下面是大多数编程语言中循环语句的流程图：



循环类型

Swift 语言提供了以下几种循环类型。点击链接查看每个类型的详细描述：

循环类型	描述
for-in	遍历一个集合里面的所有元素，例如由数字表示的区间、数组中的元素、字符串中的字符。
for 循环	用来重复执行一系列语句直到达成特定条件达成，一般通过在每次循环完成后增加计数器的值来实现。
while 循环	运行一系列语句，如果条件为true，会重复运行，直到条件变为false。
repeat...while 循环	类似 while 语句区别在于判断循环条件之前，先执行一次循环的代码块。

循环控制语句

循环控制语句改变你代码的执行顺序，通过它你可以实现代码的跳转。Swift 以下几种循环控制语句：

控制语句	描述
continue 语句	告诉一个循环体立刻停止本次循环迭代，重新开始下次循环迭代。
break 语句	中断当前循环。
fallthrough 语句	如果在一个case执行完后，继续执行下面的case，需要使用fallthrough(贯穿)关键字。

Swift for-in 循环

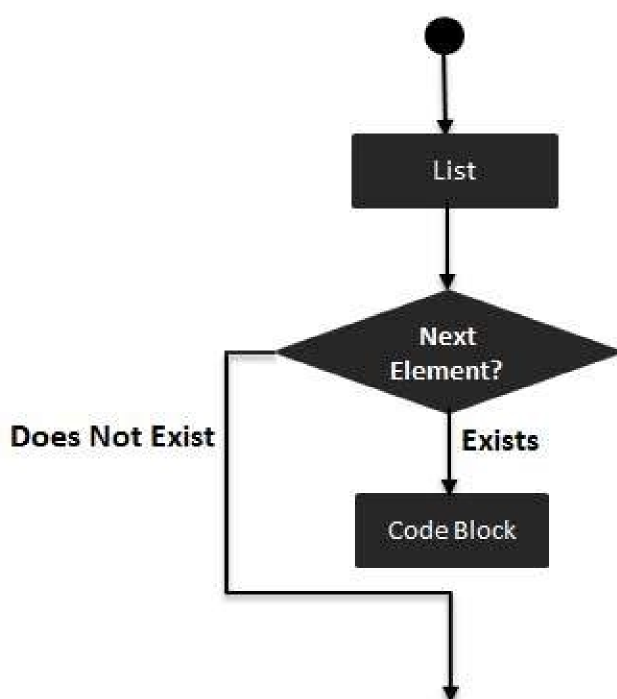
Swift for-in 循环用于遍历一个集合里面的所有元素，例如由数字表示的区间、数组中的元素、字符串中的字符。

语法

Swift for-in 循环的语法格式如下：

```
for index in var {
   循环体
}
```

流程图：



实例

```
import Cocoa

var someInts:[Int] = [10, 20, 30]

for index in someInts {
   print( "index 的值为 \(index)")
}
```

以上程序执行输出结果为：

```
index 的值为  10
index 的值为  20
index 的值为  30
```

Swift for 循环

Swift for 循环用来重复执行一系列语句直到达成特定条件，一般通过在每次循环完成后增加计数器的值来实现。

语法

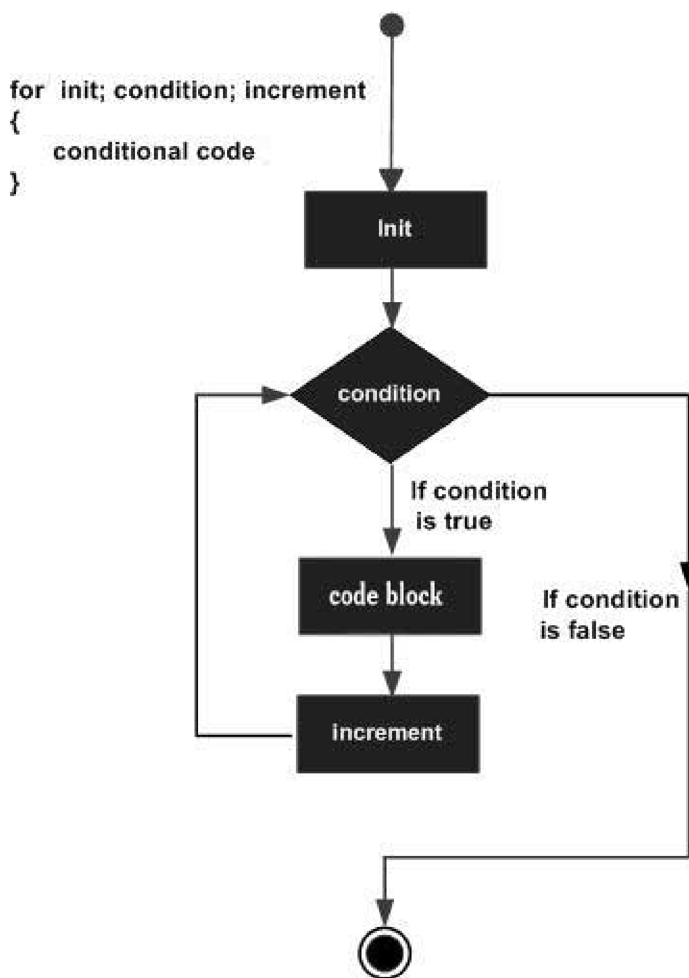
Swift for 循环的语法格式如下：

```
for init; condition; increment{
   循环体
}
```

参数解析：

1. **init** 会首先被执行，且只会执行一次。这一步允许您声明并初始化任何循环控制变量。您也可以不在这里写任何语句，只要有一个分号出现即可。
2. 接下来，会判断 **condition**。如果为真，则执行循环主体。如果为假，则不执行循环主体，且控制流会跳转到紧接着 for 循环的下一条语句。
3. 在执行完 for 循环主体后，控制流会跳回上面的 **increment** 语句。该语句允许您更新循环控制变量。该语句可以留空，只要在条件后有一个分号出现即可。
4. 条件再次被判断。如果为真，则执行循环，这个过程会不断重复（循环主体，然后增加步值，再然后重新判断条件）。在条件变为假时，for 循环终止。

流程图：



实例

```
import Cocoa

var someInts:[Int] = [10, 20, 30]

for var index = 0; index < 3; ++index {
   print( "索引 [\(index)] 对应的值为 \(someInts[index])")
}
```

以上程序执行输出结果为：

```
索引 [0] 对应的值为 10
索引 [1] 对应的值为 20
索引 [2] 对应的值为 30
```

Swift While 循环

Swift while循环从计算单一条件开始。如果条件为true，会重复运行一系列语句，直到条件变为false。

语法

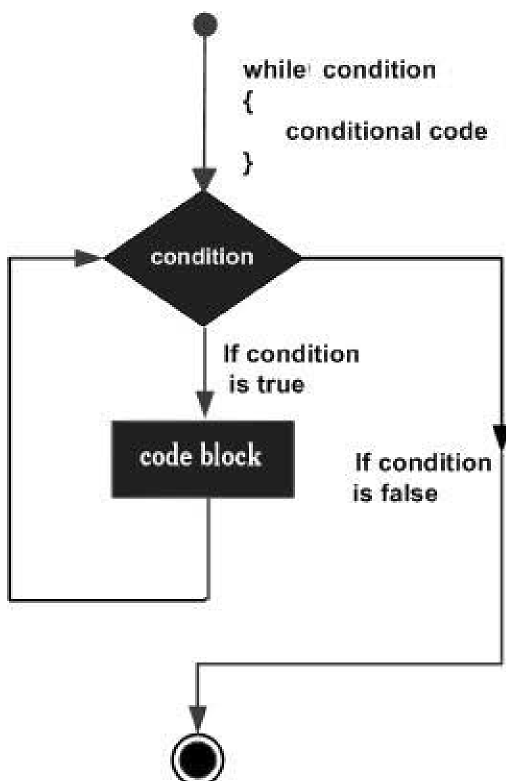
Swift while 循环的语法格式如下：

```
while condition
{
   statement(s)
}
```

语法中的 **statement(s)** 可以是一个语句或者一个语句块。 **condition** 可以是一个表达式。如果条件为true，会重复运行一系列语句，直到条件变为false。

数字 0, 字符串 '0' 和 "", 空的 list(), 及未定义的变量都为 **false**， 其他的则都为 **true**。true 取反使用 **!** 号或 **not**，取反后返回 false。

流程图：



实例

```
import Cocoa

var index = 10

while index < 20
{
   print( "index 的值为 \(index)")
   index = index + 1
}
```

以上程序执行输出结果为：

```
index 的值为 10
index 的值为 11
index 的值为 12
index 的值为 13
index 的值为 14
index 的值为 15
index 的值为 16
index 的值为 17
index 的值为 18
index 的值为 19
```

Swift repeat...while 循环

Swift repeat...while 循环不像 for 和 while 循环在循环体开始执行前先判断条件语句，而是在循环执行结束时判断条件是否符合。

语法

Swift repeat...while 循环的语法格式如下：

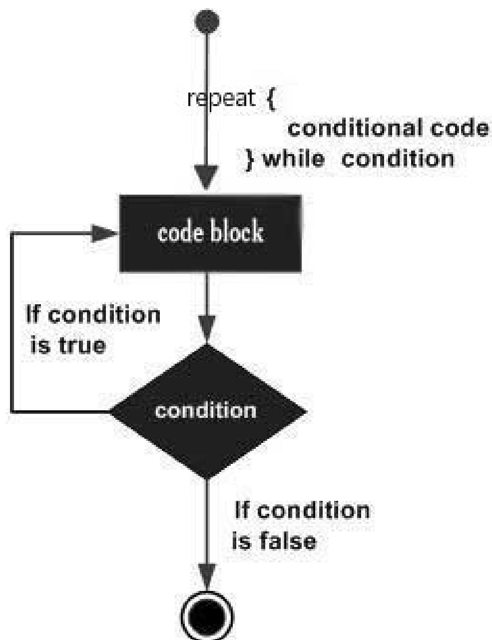
```
repeat
{
   statement(s);
}while( condition );
```

请注意，条件表达式出现在循环的尾部，所以循环中的 statement(s) 会在条件被测试之前至少执行一次。

如果条件为 true，控制流会跳转回上面的 repeat，然后重新执行循环中的 statement(s)。这个过程会不断重复，直到给定条件变为 false 为止。

数字 0, 字符串 '0' 和 "", 空的 list(), 及未定义的变量都为 **false**， 其他的则都为 **true**。true 取反使用 **!** 号或 **not**，取反后返回 false。

流程图：



实例

```
import Cocoa

var index = 15

repeat{
    print( "index 的值为 \(index)")
    index = index + 1
}while index < 20
```

以上程序执行输出结果为：

```
index 的值为  15
index 的值为  16
index 的值为  17
index 的值为  18
index 的值为  19
```

Swift Continue 语句

Swift continue语句告诉一个循环体立刻停止本次循环迭代，重新开始下次循环迭代。

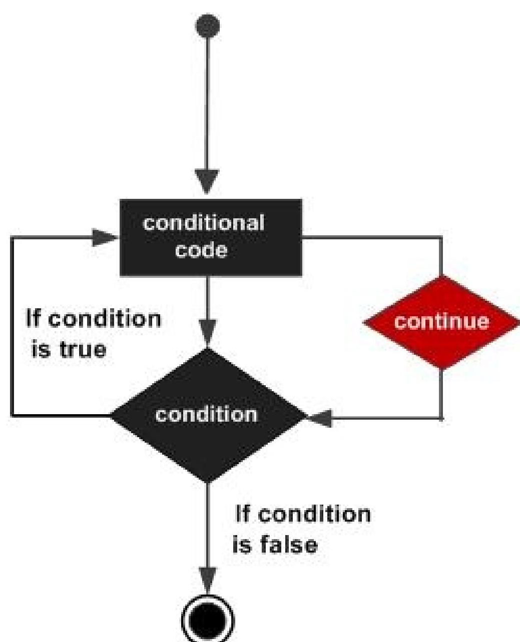
对于 **for** 循环，**continue** 语句执行后自增语句仍然会执行。对于 **while** 和 **do...while** 循环，**continue** 语句重新执行条件判断语句。

语法

Swift continue语句的语法格式如下：

```
continue
```

流程图：



实例

```
import Cocoa

var index = 10

repeat{
   index = index + 1

   if( index == 15 ){ // index 等于 15 时跳过
      continue
   }
   print( "index 的值为 \(index)")
}while index < 20
```

以上程序执行输出结果为：

```
index 的值为 11
index 的值为 12
index 的值为 13
index 的值为 14
index 的值为 16
index 的值为 17
index 的值为 18
index 的值为 19
index 的值为 20
```

Swift Break 语句

Swift break语句会立刻结束整个控制流的执行。

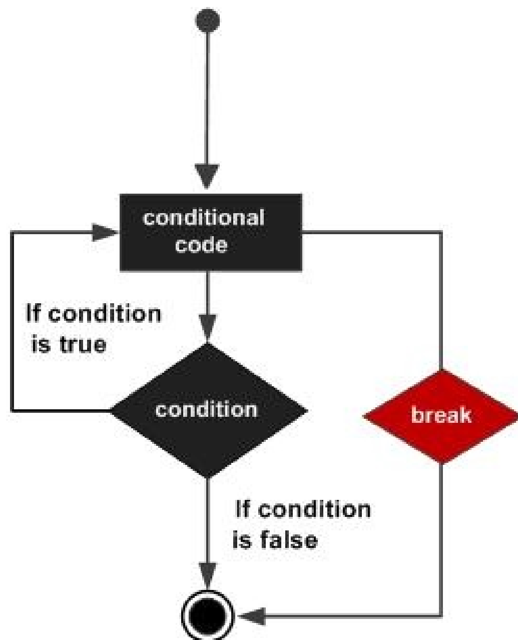
如果您使用的是嵌套循环（即一个循环内嵌套另一个循环），break 语句会停止执行最内层的循环，然后开始执行该块之后的下一行代码。

语法

Swift break语句的语法格式如下：

```
break
```

流程图：



实例

```
import Cocoa

var index = 10

repeat{
    index = index + 1

    if( index == 15 ){  // index 等于 15 时终止循环
        break
    }
    print( "index 的值为 \(index)")
}while index < 20
```

以上程序执行输出结果为：

```
index 的值为 11
index 的值为 12
index 的值为 13
index 的值为 14
```

Swift Fallthrough 语句

Swift fallthrough 语句让 case 之后的语句会按顺序继续运行，且不论条件是否满足都会执行。

Swift 中的 switch 不会从上一个 case 分支落入到下一个 case 分支中。只要第一个匹配到的 case 分支完成了它需要执行的语句，整个switch代码块完成了它的执行。

> 注意：在大多数语言中，switch 语句块中，case 要紧跟 break，否则 case 之后的语句会顺序运行，而在 Swift 语言中，默认是不会执行下去的，switch 也会终止。如果你想在 Swift 中让 case 之后的语句会按顺序继续运行，则需要使用 fallthrough 语句。

语法

Swift fallthrough 语句的语法格式如下：

```
fallthrough
```

一般在 switch 语句中不使用 fallthrough 语句。

实例1

以下实例没有使用 fallthrough 语句：

```
import Cocoa

var index = 10

switch index {
   case 100  :
      print( "index 的值为 100")
   case 10,15  :
      print( "index 的值为 10 或 15")
   case 5  :
      print( "index 的值为 5")
   default :
      print( "默认 case")
}
```

当上面的代码被编译执行时，它会产生下列结果：

```
index 的值为  10  或  15
```

实例2

以下实例使用 fallthrough 语句：

```
import Cocoa

var index = 10

switch index {
   case 100  :
      print( "index 的值为 100")
      fallthrough
   case 10,15  :
      print( "index 的值为 10 或 15")
      fallthrough
   case 5  :
      print( "index 的值为 5")
   default :
      print( "默认 case")
}
```

当上面的代码被编译执行时，它会产生下列结果：

```
index 的值为  10  或  15
index 的值为  5
```

Swift 字符串

Swift 字符串是一系列字符的集合。例如 "Hello, World!" 这样的有序的字符类型的值的集合，它的数据类型为 **String**。

创建字符串

你可以通过使用字符串字面量或 String 类的实例来创建一个字符串：

```
import Cocoa

// 使用字符串字面量
var stringA = "Hello, World!"
print( stringA )

// String 实例化
var stringB = String("Hello, World!")
print( stringB )
```

以上程序执行输出结果为：

```
Hello, World!
Hello, World!
```

空字符串

你可以使用空的字符串字面量赋值给变量或初始化一个String类的实例来初始值一个空的字符串。 我们可以使用字符串属性 isEmpty 来判断字符串是否为空：

```
import Cocoa

// 使用字符串字面量创建空字符串
var stringA = ""

if stringA.isEmpty {
   print( "stringA 是空的" )
} else {
   print( "stringA 不是空的" )
}

// 实例化 String 类来创建空字符串
let stringB = String()

if stringB.isEmpty {
   print( "stringB 是空的" )
} else {
   print( "stringB 不是空的" )
}
```

以上程序执行输出结果为：

```
stringA 是空的
stringB 是空的
```

字符串常量

你可以将一个字符串赋值给一个变量或常量，变量是可修改的，常量是不可修改的。

```
import Cocoa

// stringA 可被修改
var stringA = "菜鸟教程："
stringA += "http://www.runoob.com"
print( stringA )

// stringB 不能修改
let stringB = String("菜鸟教程：")
stringB += "http://www.runoob.com"
print( stringB )
```

以上程序执行输出结果会报错，以为 stringB 为常量是不能被修改的：

```
error: left side of mutating operator isn't mutable: 'stringB' is a
stringB += "http://www.runoob.com"
```

字符串中插入值

字符串插值是一种构建新字符串的方式，可以在其中包含常量、变量、字面量和表达式。 您插入的字符串字面量的每一项都在以反斜线为前缀的圆括号中：

```
import Cocoa

var varA   = 20
let constA = 100
var varC:Float = 20.0

var stringA = "\(varA) 乘于 \(constA) 等于 \(varC * 100)"
print( stringA )
```

以上程序执行输出结果为：

```
20 乘于 100 等于 2000.0
```

字符串连接

字符串可以通过 **+** 号来连接，实例如下：

```
import Cocoa

let constA = "菜鸟教程："
let constB = "http://www.runoob.com"

var stringA = constA + constB

print( stringA )
```

以上程序执行输出结果为：

```
菜鸟教程：http://www.runoob.com
```

字符串长度

字符串长度使用 **String.characters.count** 属性来计算，实例如下：

```
import Cocoa

var varA   = "www.runoob.com"

print( "\(varA), 长度为 \(varA.characters.count)" )
```

以上程序执行输出结果为：

```
www.runoob.com, 长度为 14
```

字符串比较

你可以使用 **==** 来比较两个字符串是否相等：

```
import Cocoa

var varA   = "Hello, Swift!"
var varB   = "Hello, World!"

if varA == varB {
   print( "\(varA) 与 \(varB) 是相等的" )
} else {
   print( "\(varA) 与 \(varB) 是不相等的" )
}
```

以上程序执行输出结果为：

```
Hello, Swift! 与 Hello, World! 是不相等的
```

Unicode 字符串

Unicode 是一个国际标准，用于文本的编码，Swift 的 String 类型是基于 Unicode 建立的。你可以循环迭代出字符串中 UTF-8 与 UTF-16 的编码，实例如下：

```
import Cocoa

var unicodeString   = "菜鸟教程"

print("UTF-8 编码: ")
for code in unicodeString.utf8 {
   print("\(code) ")
}

print("\n")

print("UTF-16 编码: ")
for code in unicodeString.utf16 {
   print("\(code) ")
}
```

以上程序执行输出结果为：

```
UTF-8 编码:
232
143
156
233
184
159
230
149
153
231
168
139
UTF-16 编码:
33756
40479
25945
31243
```

字符串函数及运算符

Swift 支持以下几种字符串函数及运算符：

函数/运算符	描述
isEmpty	判断字符串是否为空，返回布尔值
hasPrefix(prefix: String)	检查字符串是否拥有特定后缀
hasSuffix(suffix: String)	检查字符串是否拥有特定后缀。

Int(String)	转换字符串数字为整型。 实例:
	`let myString: String = "256"`
	`let myInt: Int? = Int(myString)`

String.characters.count	计算字符串的长度
utf8	您可以通过遍历 String 的 utf8 属性来访问它的 UTF-8 编码
utf16	您可以通过遍历 String 的 utf8 属性来访问它的 UTF-16 编码
unicodeScalars	您可以通过遍历String值的unicodeScalars属性来访问它的 Unicode 标量编码.
+	连接两个字符串，并返回一个新的字符串
+=	连接操作符两边的字符串并将新字符串赋值给左边的操作符变量
==	判断两个字符串是否相等
<	比较两个字符串，对两个字符串的字母逐一比较。
!=	比较两个字符串是否不相等。

Swift 字符(Character)

Swift 的字符是一个单一的字符字符串字面量，数据类型为 Character。

以下实例列出了两个字符实例：

```
import Cocoa

let char1: Character = "A"
let char2: Character = "B"

print("char1 的值为 \(char1)")
print("char2 的值为 \(char2)")
```

以上程序执行输出结果为：

```
char1 的值为 A
char2 的值为 B
```

如果你想在 Character（字符） 类型的常量中存储更多的字符，则程序执行会报错，如下所示：

```
import Cocoa

// Swift 中以下赋值会报错
let char: Character = "AB"

print("Value of char \(char)")
```

以上程序执行输出结果为：

```
error: cannot convert value of type 'String' to specified type 'Cha
let char: Character = "AB"
```

空字符变量

Swift 中不能创建空的 Character（字符） 类型变量或常量：

```
import Cocoa

// Swift 中以下赋值会报错
let char1: Character = ""
var char2: Character = ""

print("char1 的值为 \(char1)")
print("char2 的值为 \(char2)")
```

以上程序执行输出结果为：

```
 error: cannot convert value of type 'String' to specified type 'Ch
let char1: Character = ""
                       ^~
error: cannot convert value of type 'String' to specified type 'Cha
var char2: Character = ""
```

遍历字符串中的字符

Swift 的 String 类型表示特定序列的 Character（字符） 类型值的集合。 每一个字符值代表一个 Unicode 字符。

您可通过for-in循环来遍历字符串中的characters属性来获取每一个字符的值：

```
import Cocoa

for ch in "Hello".characters {
   print(ch)
}
```

以上程序执行输出结果为：

```
H
e
l
l
o
```

字符串连接字符

以下实例演示了使用 String 的 append() 方法来实现字符串连接字符：

```
import Cocoa

var varA:String = "Hello "
let varB:Character = "G"

varA.append( varB )

print("varC  =  \(varA)")
```

以上程序执行输出结果为：

```
varC  =  Hello G
```

Swift 数组

Swift 数组使用有序列表存储同一类型的多个值。相同的值可以多次出现在一个数组的不同位置中。

Swift 数组会强制检测元素的类型，如果类型不同则会报错，Swift 数组应该遵循像Array<Element>这样的形式，其中Element是这个数组中唯一允许存在的数据类型。

如果创建一个数组，并赋值给一个变量，则创建的集合就是可以修改的。这意味着在创建数组后，可以通过添加、删除、修改的方式改变数组里的项目。如果将一个数组赋值给常量，数组就不可更改，并且数组的大小和内容都不可以修改。

创建数组

我们可以使用构造语法来创建一个由特定数据类型构成的空数组：

```
var someArray = [SomeType]()
```

以下是创建一个初始化大小数组的语法：

```
var someArray = [SomeType](count: NumbeOfElements, repeatedValue: I
```

以下实例创建了一个类型为 Int ，大小为 3，初始值为 0 的空数组：

```
var someInts = [Int](count: 3, repeatedValue: 0)
```

以下实例创建了含有三个元素的数组：

```
var someInts:[Int] = [10, 20, 30]
```

访问数组

我们可以根据数组的索引来访问数组的元素，语法如下：

```
var someVar = someArray[index]
```

index 索引从 0 开始，及索引 0 对应第一个元素，索引 1 对应第二个元素，以此类推。

我们可以通过以下实例来学习如何创建，初始化，访问数组：

```
import Cocoa

var someInts = [Int](count: 3, repeatedValue: 10)

var someVar = someInts[0]

print( "第一个元素的值 \(someVar)" )
print( "第二个元素的值 \(someInts[1])" )
print( "第三个元素的值 \(someInts[2])" )
```

以上程序执行输出结果为：

```
第一个元素的值 10
第二个元素的值 10
第三个元素的值 10
```

修改数组

你可以使用 append() 方法或者赋值运算符 += 在数组末尾添加元素，如下所示，我们初始化一个数组，并向其添加元素：

```
import Cocoa

var someInts = [Int]()

someInts.append(20)
someInts.append(30)
someInts += [40]

var someVar = someInts[0]

print( "第一个元素的值 \(someVar)" )
print( "第二个元素的值 \(someInts[1])" )
print( "第三个元素的值 \(someInts[2])" )
```

以上程序执行输出结果为：

```
第一个元素的值 20
第二个元素的值 30
第三个元素的值 40
```

我们也可以通过索引修改数组元素的值：

```
import Cocoa

var someInts = [Int]()

someInts.append(20)
someInts.append(30)
someInts += [40]

// 修改最后一个元素
someInts[2] = 50

var someVar = someInts[0]

print( "第一个元素的值 \(someVar)" )
print( "第二个元素的值 \(someInts[1])" )
print( "第三个元素的值 \(someInts[2])" )
```

以上程序执行输出结果为：

```
第一个元素的值 20
第二个元素的值 30
第三个元素的值 50
```

遍历数组

我们可以使用for-in循环来遍历所有数组中的数据项：

```
import Cocoa

var someStrs = [String]()

someStrs.append("Apple")
someStrs.append("Amazon")
someStrs.append("Runoob")
someStrs += ["Google"]

for item in someStrs {
   print(item)
}
```

以上程序执行输出结果为：

```
Apple
Amazon
Runoob
Google
```

如果我们同时需要每个数据项的值和索引值，可以使用 String 的 enumerate() 方法来进行数组遍历。实例如下：

```
import Cocoa

var someStrs = [String]()

someStrs.append("Apple")
someStrs.append("Amazon")
someStrs.append("Runoob")
someStrs += ["Google"]

for (index, item) in someStrs.enumerate() {
   print("在 index = \(index) 位置上的值为 \(item)")
}
```

以上程序执行输出结果为：

```
在 index = 0 位置上的值为 Apple
在 index = 1 位置上的值为 Amazon
在 index = 2 位置上的值为 Runoob
在 index = 3 位置上的值为 Google
```

合并数组

我们可以使用加法操作符（+）来合并两种已存在的相同类型数组。新数组的数据类型会从两个数组的数据类型中推断出来：

```
import Cocoa

var intsA = [Int](count:2, repeatedValue: 2)
var intsB = [Int](count:3, repeatedValue: 1)

var intsC = intsA + intsB

for item in intsC {
   print(item)
}
```

以上程序执行输出结果为：

```
2
2
1
1
1
```

count 属性

我们可以使用 count 属性来计算数组元素个数：

```
import Cocoa

var intsA = [Int](count:2, repeatedValue: 2)
var intsB = [Int](count:3, repeatedValue: 1)

var intsC = intsA + intsB

print("intsA 元素个数为 \(intsA.count)")
print("intsB 元素个数为 \(intsB.count)")
print("intsC 元素个数为 \(intsC.count)")
```

以上程序执行输出结果为：

```
intsA 元素个数为 2
intsB 元素个数为 3
intsC 元素个数为 5
```

isEmpty 属性

我们可以通过只读属性 isEmpty 来判断数组是否为空，返回布尔值：

```
import Cocoa

var intsA = [Int](count:2, repeatedValue: 2)
var intsB = [Int](count:3, repeatedValue: 1)
var intsC = [Int]()

print("intsA.isEmpty = \(intsA.isEmpty)")
print("intsB.isEmpty = \(intsB.isEmpty)")
print("intsC.isEmpty = \(intsC.isEmpty)")
```

以上程序执行输出结果为：

```
intsA.isEmpty = false
intsB.isEmpty = false
intsC.isEmpty = true
```

Swift 字典

Swift 字典用来存储无序的相同类型数据的集合，Swift 数组会强制检测元素的类型，如果类型不同则会报错。

Swift 字典每个值（value）都关联唯一的键（key），键作为字典中的这个值数据的标识符。

和数组中的数据项不同，字典中的数据项并没有具体顺序。我们在需要通过标识符（键）访问数据的时候使用字典，这种方法很大程度上和我们在现实世界中使用字典查字义的方法一样。

Swift 字典的key没有类型限制可以是整型或字符串，但必须是唯一的。

如果创建一个字典，并赋值给一个变量，则创建的字典就是可以修改的。这意味着在创建字典后，可以通过添加、删除、修改的方式改变字典里的项目。如果将一个字典赋值给常量，字典就不可修改，并且字典的大小和内容都不可以修改。

创建字典

我们可以使用以下语法来创建一个特定类型的空字典：

```
var someDict =  [KeyType: ValueType]()
```

以下是创建一个空字典，键的类型为 Int，值的类型为 String 的简单语法：

```
var someDict = [Int: String]()
```

以下为创建一个字典的实例：

```
var someDict:[Int:String] = [1:"One", 2:"Two", 3:"Three"]
```

访问字典

我们可以根据字典的索引来访问数组的元素，语法如下：

```
var someVar = someDict[key]
```

我们可以通过以下实例来学习如何创建，初始化，访问字典:

```
import Cocoa

var someDict:[Int:String] = [1:"One", 2:"Two", 3:"Three"]

var someVar = someDict[1]

print( "key = 1 的值为 \(someVar)" )
print( "key = 2 的值为 \(someDict[2])" )
print( "key = 3 的值为 \(someDict[3])" )
```

以上程序执行输出结果为：

```
key = 1 的值为 Optional("One")
key = 2 的值为 Optional("Two")
key = 3 的值为 Optional("Three")
```

修改字典

我们可以使用 **updateValue(forKey:)** 增加或更新字典的内容。如果 key 不存在，则添加值，如果存在则修改 key 对应的值。updateValue(_:forKey:)方法返回Optional值。实例如下：

```
import Cocoa

var someDict:[Int:String] = [1:"One", 2:"Two", 3:"Three"]

var oldVal = someDict.updateValue("One 新的值", forKey: 1)

var someVar = someDict[1]

print( "key = 1 旧的值 \(oldVal)" )
print( "key = 1 的值为 \(someVar)" )
print( "key = 2 的值为 \(someDict[2])" )
print( "key = 3 的值为 \(someDict[3])" )
```

以上程序执行输出结果为：

```
key = 1 旧的值 Optional("One")
key = 1 的值为 Optional("One 新的值")
key = 2 的值为 Optional("Two")
key = 3 的值为 Optional("Three")
```

你也可以通过指定的 key 来修改字典的值，如下所示：

```
import Cocoa

var someDict:[Int:String] = [1:"One", 2:"Two", 3:"Three"]

var oldVal = someDict[1]
someDict[1] = "One 新的值"
var someVar = someDict[1]

print( "key = 1 旧的值 \(oldVal)" )
print( "key = 1 的值为 \(someVar)" )
print( "key = 2 的值为 \(someDict[2])" )
print( "key = 3 的值为 \(someDict[3])" )
```

以上程序执行输出结果为：

```
key = 1 旧的值 Optional("One")
key = 1 的值为 Optional("One 新的值")
key = 2 的值为 Optional("Two")
key = 3 的值为 Optional("Three")
```

移除 Key-Value 对

我们可以使用 **removeValueForKey()** 方法来移除字典 key-value 对。如果 key 存在该方法返回移除的值，如果不存在返回 nil 。实例如下：

```
import Cocoa

var someDict:[Int:String] = [1:"One", 2:"Two", 3:"Three"]

var removedValue = someDict.removeValueForKey(2)

print( "key = 1 的值为 \(someDict[1])" )
print( "key = 2 的值为 \(someDict[2])" )
print( "key = 3 的值为 \(someDict[3])" )
```

以上程序执行输出结果为：

```
key = 1 的值为 Optional("One")
key = 2 的值为 nil
key = 3 的值为 Optional("Three")
```

你也可以通过指定键的值为 nil 来移除 key-value（键-值）对。实例如下:

```
import Cocoa

var someDict:[Int:String] = [1:"One", 2:"Two", 3:"Three"]

someDict[2] = nil

print( "key = 1 的值为 \(someDict[1])" )
print( "key = 2 的值为 \(someDict[2])" )
print( "key = 3 的值为 \(someDict[3])" )
```

以上程序执行输出结果为：

```
key = 1 的值为 Optional("One")
key = 2 的值为 nil
key = 3 的值为 Optional("Three")
```

遍历字典

我们可以使用 **for-in** 循环来遍历某个字典中的键值对。实例如下:

```
import Cocoa

var someDict:[Int:String] = [1:"One", 2:"Two", 3:"Three"]

for (key, value) in someDict {
   print("字典 key \(key) -  字典 value \(value)")
}
```

以上程序执行输出结果为：

```
字典 key 2 -  字典 value Two
字典 key 3 -  字典 value Three
字典 key 1 -  字典 value One
```

我们也可以使用enumerate()方法来进行字典遍历，返回的是字典的索引及 (key, value) 对，实例如下:

```
import Cocoa

var someDict:[Int:String] = [1:"One", 2:"Two", 3:"Three"]

for (key, value) in someDict.enumerate() {
    print("字典 key \(key) -  字典 (key, value) 对 \(value)")
}
```

以上程序执行输出结果为：

```
字典 key 0 -  字典 (key, value) 对 (2, "Two")
字典 key 1 -  字典 (key, value) 对 (3, "Three")
字典 key 2 -  字典 (key, value) 对 (1, "One")
```

字典转换为数组

你可以提取字典的键值(key-value)对，并转换为独立的数组。实例如下：

```
import Cocoa

var someDict:[Int:String] = [1:"One", 2:"Two", 3:"Three"]

let dictKeys = [Int](someDict.keys)
let dictValues = [String](someDict.values)

print("输出字典的键(key)")

for (key) in dictKeys {
    print("\(key)")
}

print("输出字典的值(value)")

for (value) in dictValues {
    print("\(value)")
}
```

以上程序执行输出结果为：

```
输出字典的键(key)
2
3
1
输出字典的值(value)
Two
Three
One
```

count 属性

我们可以使用只读的 **count** 属性来计算字典有多少个键值对：

```
import Cocoa

var someDict1:[Int:String] = [1:"One", 2:"Two", 3:"Three"]
var someDict2:[Int:String] = [4:"Four", 5:"Five"]

print("someDict1 含有 \(someDict1.count) 个键值对")
print("someDict2 含有 \(someDict2.count) 个键值对")
```

以上程序执行输出结果为：

```
someDict1 含有 3 个键值对
someDict2 含有 2 个键值对
```

isEmpty 属性

Y我们可以通过只读属性 **isEmpty** 来判断字典是否为空，返回布尔值:

```
import Cocoa

var someDict1:[Int:String] = [1:"One", 2:"Two", 3:"Three"]
var someDict2:[Int:String] = [4:"Four", 5:"Five"]
var someDict3:[Int:String] = [Int:String]()

print("someDict1 = \(someDict1.isEmpty)")
print("someDict2 = \(someDict2.isEmpty)")
print("someDict3 = \(someDict3.isEmpty)")
```

以上程序执行输出结果为：

```
someDict1 = false
someDict2 = false
someDict3 = true
```

Swift 函数

Swift 函数用来完成特定任务的独立的代码块。

Swift使用一个统一的语法来表示简单的C语言风格的函数到复杂的Objective-C语言风格的方法。

- 函数声明: 告诉编译器函数的名字，返回类型及参数。
- 函数定义: 提供了函数的实体。

Swift 函数包含了参数类型及返回值类型：

函数定义

Swift 定义函数使用关键字 **func**。

定义函数的时候，可以指定一个或多个输入参数和一个返回值类型。

每个函数都有一个函数名来描述它的功能。通过函数名以及对应类型的参数值来调用这个函数。函数的参数传递的顺序必须与参数列表相同。

函数的实参传递的顺序必须与形参列表相同，**->** 后定义函数的返回值类型。

语法

```
func funcname(形参) -> returntype
{
   Statement1
   Statement2
   ……
   Statement N
   return parameters
}
```

实例

以下我们定义了一个函数名为 runoob 的函数，形参的数据类型为 String，返回值也为 String：

```
import Cocoa

func runoob(site: String) -> String {
    return site
}
print(runoob("www.runoob.com"))
```

以上程序执行输出结果为：

```
www.runoob.com
```

函数调用

我们可以通过函数名以及对应类型的参数值来调用函数，函数的参数传递的顺序必须与参数列表相同。

以下我们定义了一个函数名为 runoob 的函数，形参 site 的数据类型为 String，之后我们调用函数传递的实参也必须 String 类型，实参传入函数体后，将直接返回，返回的数据类型为 String。

```
import Cocoa

func runoob(site: String) -> String {
    return site
}
print(runoob("www.runoob.com"))
```

以上程序执行输出结果为：

```
www.runoob.com
```

函数参数

函数可以接受一个或者多个参数，我们也可以使用元组（tuple）向函数传递一个或多个参数：

```
import Cocoa

func mult(no1: Int, no2: Int) -> Int {
   return no1*no2
}
print(mult(2, no2:20))
print(mult(3, no2:15))
print(mult(4, no2:30))
```

以上程序执行输出结果为：

```
40
45
120
```

不带参数函数

我们可以创建不带参数的函数。

语法：

```
func funcname() -> datatype {
   return datatype
}
```

实例

```
import Cocoa

func sitename() -> String {
    return "菜鸟教程"
}
print(sitename())
```

以上程序执行输出结果为：

```
菜鸟教程
```

元组作为函数返回值

函数返回值类型可以是字符串，整型，浮点型等。

元组与数组类似，不同的是，元组中的元素可以是任意类型，使用的是圆括号。

你可以用元组（tuple）类型让多个值作为一个复合值从函数中返回。

下面的这个例子中，定义了一个名为minMax(_:)的函数，作用是在一个Int数组中找出最小值与最大值。

```
import Cocoa

func minMax(array: [Int]) -> (min: Int, max: Int) {
    var currentMin = array[0]
    var currentMax = array[0]
    for value in array[1..<array.count] {
        if value < currentMin {
            currentMin = value
        } else if value > currentMax {
            currentMax = value
        }
    }
    return (currentMin, currentMax)
}

let bounds = minMax([8, -6, 2, 109, 3, 71])
print("最小值为 \(bounds.min) ，最大值为 \(bounds.max)")
```

minMax(_:)函数返回一个包含两个Int值的元组，这些值被标记为min和max，以便查询函数的返回值时可以通过名字访问它们。

以上程序执行输出结果为：

```
最小值为 -6 ，最大值为 109
```

如果你不确定返回的元组一定不为nil，那么你可以返回一个可选的元组类型。

你可以通过在元组类型的右括号后放置一个问号来定义一个可选元组，例如(Int, Int)?或(String, Int, Bool)?

> 注意 可选元组类型如 `(Int, Int)?` 与元组包含可选类型如 `(Int?, Int?)` 是不同的.可选的元组类型，整个元组是可选的，而不只是元组中的每个元素值。

前面的 `minMax(_:)` 函数返回了一个包含两个 `Int` 值的元组。但是函数不会对传入的数组执行任何安全检查，如果 `array` 参数是一个空数组，如上定义的 `minMax(_:)` 在试图访问 `array[0]` 时会触发一个运行时错误。

为了安全地处理这个"空数组"问题，将 `minMax(_:)` 函数改写为使用可选元组返回类型，并且当数组为空时返回 `nil` ：

```
import Cocoa

func minMax(array: [Int]) -> (min: Int, max: Int)? {
    if array.isEmpty { return nil }
    var currentMin = array[0]
    var currentMax = array[0]
    for value in array[1..<array.count] {
        if value < currentMin {
            currentMin = value
        } else if value > currentMax {
            currentMax = value
        }
    }
    return (currentMin, currentMax)
}
if let bounds = minMax([8, -6, 2, 109, 3, 71]) {
    print("最小值为 \(bounds.min)，组大值为 \(bounds.max)")
}
```

以上程序执行输出结果为：

```
最小值为 -6，组大值为 109
```

没有返回值函数

下面是 runoob(_:) 函数的另一个版本，这个函数接收菜鸟教程官网网址参数，没有指定返回值类型，并直接输出 String 值，而不是返回它：

```
import Cocoa

func runoob(site: String) {
    print("菜鸟教程官网：\(site)")
}
runoob("http://www.runoob.com")
```

以上程序执行输出结果为：

```
菜鸟教程官网：http://www.runoob.com
```

函数参数名称

函数参数都有一个外部参数名和一个局部参数名。

局部参数名

局部参数名在函数的实现内部使用。

```
func sample(number: Int) {
   println(number)
}
```

以上实例中 number 为局部参数名，只能在函数体内使用。

```
import Cocoa

func sample(number: Int) {
   print(number)
}
sample(1)
sample(2)
sample(3)
```

以上程序执行输出结果为：

```
1
2
3
```

外部参数名

你可以在局部参数名前指定外部参数名，中间以空格分隔，外部参数名用于在函数调用时传递给函数的参数。

如下你可以定义以下两个函数参数名并调用它：

```
import Cocoa

func pow(firstArg a: Int, secondArg b: Int) -> Int {
   var res = a
   for _ in 1..<b {
      res = res * a
   }
   print(res)
   return res
}
pow(firstArg:5, secondArg:3)
```

以上程序执行输出结果为：

```
125
```

> 注意 如果你提供了外部参数名，那么函数在被调用时，必须使用外部参数名。

可变参数

可变参数可以接受零个或多个值。函数调用时，你可以用可变参数来指定函数参数，其数量是不确定的。

可变参数通过在变量类型名后面加入（...）的方式来定义。

```
import Cocoa

func vari<N>(members: N...){
    for i in members {
        print(i)
    }
}
vari(4,3,5)
vari(4.5, 3.1, 5.6)
vari("Google", "Baidu", "Runoob")
```

以上程序执行输出结果为：

```
4
3
5
4.5
3.1
5.6
Google
Baidu
Runoob
```

常量，变量及 I/O 参数

一般默认在函数中定义的参数都是常量参数，也就是这个参数你只可以查询使用，不能改变它的值。

如果想要声明一个变量参数，可以在前面加上var，这样就可以改变这个参数的值了。

例如：

```
func  getName(var id:String).........
```

此时这个id值可以在函数中改变。

一般默认的参数传递都是传值调用的，而不是传引用。 所以传入的参数在函数内改变，并不影响原来的那个参数。传入的只是这个参数的副本。

变量参数，正如上面所述，仅仅能在函数体内被更改。如果你想要一个函数可以修改参数的值，并且想要在这些修改在函数调用结束后仍然存在，那么就应该把这个参数定义为输入输出参数（In-Out Parameters）。

定义一个输入输出参数时，在参数定义前加 inout 关键字。一个输入输出参数有传入函数的值，这个值被函数修改，然后被传出函数，替换原来的值。

实例

```
import Cocoa

func swapTwoInts(var a:Int,var b:Int){

    let t = a
    a = b
    b = t
}

var x = 0,y = 100
print("x = \(x) ;y = \(y)")

swapTwoInts(x, b:y)
print("x = \(x) ;y = \(y)")
```

以上程序执行输出结果为：

```
x = 0 ;y = 100
x = 0 ;y = 100
```

修改方法是使用inout关键字：

```
import Cocoa

func swapTwoInts(inout a:Int,inout b:Int){

    let t = a
    a = b
    b = t
}

var x = 0,y = 100
print("x = \(x) ;y = \(y)")

swapTwoInts(&x, b:&y)
print("x = \(x) ;y = \(y)")
```

以上程序执行输出结果为：

```
x = 0 ;y = 100
x = 100 ;y = 0
```

函数类型及使用

每个函数都有种特定的函数类型，由函数的参数类型和返回类型组成。

```
func inputs(no1: Int, no2: Int) -> Int {
   return no1/no2
}
```

实例如下：

```
import Cocoa

func inputs(no1: Int, no2: Int) -> Int {
    return no1/no2
}
print(inputs(20,no2:10))
print(inputs(36,no2:6))
```

以上程序执行输出结果为：

```
2
6
```

以上函数定义了两个 Int 参数类型，返回值也为 Int 类型。

接下来我们看下如下函数，函数定义了参数为 String 类型，返回值为 String 类型。

```
Func inputstr(name: String) -> String {
   return name
}
```

函数也可以定义任何参数及类型，如下所示：

```
import Cocoa

func inputstr() {
   print("菜鸟教程")
   print("www.runoob.com")
}
inputstr()
```

以上程序执行输出结果为：

```
菜鸟教程
www.runoob.com
```

使用函数类型

在 Swift 中，使用函数类型就像使用其他类型一样。例如，你可以定义一个类型为函数的常量或变量，并将适当的函数赋值给它：

```
var addition: (Int, Int) -> Int = sum
```

解析:

"定义一个叫做 `addition` 的变量，参数与返回值类型均是 `Int` ，并让这个新变量指向 `sum` 函数"。

`sum` 和 `addition` 有同样的类型，所以以上操作是合法的。

现在，你可以用 addition 来调用被赋值的函数了：

```
import Cocoa

func sum(a: Int, b: Int) -> Int {
   return a + b
}
var addition: (Int, Int) -> Int = sum
print("输出结果: \(addition(40, 89))")
```

以上程序执行输出结果为：

```
输出结果: 129
```

函数类型作为参数类型、函数类型作为返回类型

我们可以将函数作为参数传递给另外一个参数：

```
import Cocoa

func sum(a: Int, b: Int) -> Int {
    return a + b
}
var addition: (Int, Int) -> Int = sum
print("输出结果: \(addition(40, 89))")

func another(addition: (Int, Int) -> Int, a: Int, b: Int) {
    print("输出结果: \(addition(a, b))")
}
another(sum, a: 10, b: 20)
```

以上程序执行输出结果为：

```
输出结果: 129
输出结果: 30
```

函数嵌套

函数嵌套指的是函数内定义一个新的函数，外部的函数可以调用函数内定义的函数。

实例如下：

```
import Cocoa

func calcDecrement(forDecrement total: Int) -> () -> Int {
   var overallDecrement = 0
   func decrementer() -> Int {
      overallDecrement -= total
      return overallDecrement
   }
   return decrementer
}
let decrem = calcDecrement(forDecrement: 30)
print(decrem())
```

以上程序执行输出结果为：

```
-30
```

Swift 闭包

闭包(Closures)是自包含的功能代码块，可以在代码中使用或者用来作为参数传值。

Swift 中的闭包与 C 和 Objective-C 中的代码块（blocks）以及其他一些编程语言中的 匿名函数比较相似。

全局函数和嵌套函数其实就是特殊的闭包。

闭包的形式有：

全局函数	嵌套函数	闭包表达式
有名字但不能捕获任何值。	有名字，也能捕获封闭函数内的值。	无名闭包，使用轻量级语法，可以根据上下文环境捕获值。

Swift中的闭包有很多优化的地方:

1. 根据上下文推断参数和返回值类型
2. 从单行表达式闭包中隐式返回（也就是闭包体只有一行代码，可以省略return）
3. 可以使用简化参数名，如$0, $1(从0开始，表示第i个参数...)
4. 提供了尾随闭包语法(Trailing closure syntax)

语法

以下定义了一个接收参数并返回指定类型的闭包语法：

```
{(parameters) -> return type in
   statements
}
```

实例

```
import Cocoa

let studname = { print("Swift 闭包实例。") }
studname()
```

以上程序执行输出结果为：

```
Swift 闭包实例。
```

以下闭包形式接收两个参数并返回布尔值：

```
{(Int, Int) -> Bool in
   Statement1
   Statement 2
    ---
   Statement n
}
```

实例

```
import Cocoa

let divide = {(val1: Int, val2: Int) -> Int in
   return val1 / val2
}
let result = divide(200, 20)
print (result)
```

以上程序执行输出结果为：

```
10
```

闭包表达式

闭包表达式是一种利用简洁语法构建内联闭包的方式。 闭包表达式提供了一些语法优化，使得撰写闭包变得简单明了。

sort 函数

Swift 标准库提供了名为sort的函数，会根据您提供的用于排序的闭包函数将已知类型数组中的值进行排序。

排序完成后，sort(*:)方法会返回一个与原数组大小相同,包含同类型元素且元素已正确排序的新数组，原数组不会被sort(*:)方法修改。

sort(_:)方法需要传入两个参数：

- 已知类型的数组
- 闭包函数，该闭包函数需要传入与数组元素类型相同的两个值，并返回一个布尔类型值来表明当排序结束后传入的第一个参数排在第二个参数前面还是后面。如果第一个参数值出现在第二个参数值前面，排序闭包函数需要返回 `true` ，反之返回 `false` 。

实例

```
import Cocoa

let names = ["AT", "AE", "D", "S", "BE"]

// 使用普通函数(或内嵌函数)提供排序功能,闭包函数类型需为(String, String) ->
func backwards(s1: String, s2: String) -> Bool {
    return s1 > s2
}
var reversed = names.sort(backwards)

print(reversed)
```

以上程序执行输出结果为：

```
["S", "D", "BE", "AT", "AE"]
```

如果第一个字符串 (s1) 大于第二个字符串 (s2)，backwards函数返回true，表示在新的数组中s1应该出现在s2前。 对于字符串中的字符来说，"大于" 表示 "按照字母顺序较晚出现"。 这意味着字母"B"大于字母"A"，字符串"S"大于字符串"D"。 其将进行字母逆序排序，"AT"将会排在"AE"之前。

参数名称缩写

Swift 自动为内联函数提供了参数名称缩写功能，您可以直接通过$0,$1,$2来顺序调用闭包的参数。

实例

```
import Cocoa

let names = ["AT", "AE", "D", "S", "BE"]

var reversed = names.sort( { $0 > $1 } )
print(reversed)
```

$0和$1表示闭包中第一个和第二个String类型的参数。

以上程序执行输出结果为：

```
["S", "D", "BE", "AT", "AE"]
```

如果你在闭包表达式中使用参数名称缩写, 您可以在闭包参数列表中省略对其定义, 并且对应参数名称缩写的类型会通过函数类型进行推断。in 关键字同样也可以被省略.

运算符函数

实际上还有一种更简短的方式来撰写上面例子中的闭包表达式。

Swift 的 `String` 类型定义了关于大于号 (`>`) 的字符串实现，其作为一个函数接受两个 `String` 类型的参数并返回 `Bool` 类型的值。 而这正好与 `sort(_:)` 方法的第二个参数需要的函数类型相符合。 因此，您可以简单地传递一个大于号，Swift可以自动推断出您想使用大于号的字符串函数实现：

```
import Cocoa

let names = ["AT", "AE", "D", "S", "BE"]

var reversed = names.sort(>)
print(reversed)
```

以上程序执行输出结果为：

```
["S",  "D",  "BE",  "AT",  "AE"]
```

尾随闭包

尾随闭包是一个书写在函数括号之后的闭包表达式，函数支持将其作为最后一个参数调用。

```
func someFunctionThatTakesAClosure(closure:  ()  ->  Void)  {  //
```

实例

```
func someFunctionThatTakesAClosure(closure: () -> Void) {
    // 函数体部分
}

// 以下是不使用尾随闭包进行函数调用
someFunctionThatTakesAClosure({
    // 闭包主体部分
})

// 以下是使用尾随闭包进行函数调用
someFunctionThatTakesAClosure() {
  // 闭包主体部分
}
```

sort() 后的 { $0 > $1} 为尾随闭包。

以上程序执行输出结果为：

```
["S", "D", "BE", "AT", "AE"]
```

> 注意： 如果函数只需要闭包表达式一个参数，当您使用尾随闭包时，您甚至可以把 `()` 省略掉。
>
> ```
> reversed = names.sort { $0 > $1 }
> ```

捕获值

闭包可以在其定义的上下文中捕获常量或变量。

即使定义这些常量和变量的原域已经不存在，闭包仍然可以在闭包函数体内引用和修改这些值。

Swift最简单的闭包形式是嵌套函数，也就是定义在其他函数的函数体内的函数。

嵌套函数可以捕获其外部函数所有的参数以及定义的常量和变量。

看这个例子：

```
func makeIncrementor(forIncrement amount: Int) -> () -> Int {
    var runningTotal = 0
    func incrementor() -> Int {
        runningTotal += amount
        return runningTotal
    }
    return incrementor
}
```

一个函数makeIncrementor，它有一个Int型的参数amout, 并且它有一个外部参数名字forIncremet，意味着你调用的时候，必须使用这个外部名字。返回值是一个 `()-> Int` 的函数。

函数题内，声明了变量runningTotal 和一个函数incrementor。

incrementor函数并没有获取任何参数，但是在函数体内访问了runningTotal和amount变量。这是因为其通过捕获在包含它的函数体内已经存在的runningTotal和amount变量而实现。

由于没有修改amount变量，incrementor实际上捕获并存储了该变量的一个副本，而该副本随着incrementor一同被存储。

所以我们调用这个函数时会累加：

```
import Cocoa

func makeIncrementor(forIncrement amount: Int) -> () -> Int {
    var runningTotal = 0
    func incrementor() -> Int {
        runningTotal += amount
        return runningTotal
    }
    return incrementor
}

let incrementByTen = makeIncrementor(forIncrement: 10)

// 返回的值为10
print(incrementByTen())

// 返回的值为20
print(incrementByTen())

// 返回的值为30
print(incrementByTen())
```

以上程序执行输出结果为：

```
10
20
30
```

闭包是引用类型

上面的例子中，incrementByTen是常量，但是这些常量指向的闭包仍然可以增加其捕获的变量值。

这是因为函数和闭包都是引用类型。

无论您将函数/闭包赋值给一个常量还是变量，您实际上都是将常量/变量的值设置为对应函数/闭包的引用。 上面的例子中，incrementByTen指向闭包的引用是一个常量，而并非闭包内容本身。

这也意味着如果您将闭包赋值给了两个不同的常量/变量，两个值都会指向同一个闭包：

```
import Cocoa

func makeIncrementor(forIncrement amount: Int) -> () -> Int {
    var runningTotal = 0
    func incrementor() -> Int {
        runningTotal += amount
        return runningTotal
    }
    return incrementor
}

let incrementByTen = makeIncrementor(forIncrement: 10)

// 返回的值为10
incrementByTen()

// 返回的值为20
incrementByTen()

// 返回的值为30
incrementByTen()

// 返回的值为40
incrementByTen()

let alsoIncrementByTen = incrementByTen

// 返回的值也为50
print(alsoIncrementByTen())
```

以上程序执行输出结果为：

```
50
```

Swift 枚举

枚举简单的说也是一种数据类型，只不过是这种数据类型只包含自定义的特定数据，它是一组有共同特性的数据的集合。

Swift 的枚举类似于 Objective C 和 C 的结构，枚举的功能为:

- 它声明在类中，可以通过实例化类来访问它的值。
- 枚举也可以定义构造函数（initializers）来提供一个初始成员值；可以在原始的实现基础上扩展它们的功能。
- 可以遵守协议（protocols）来提供标准的功能。

语法

Swift 中使用 enum 关键词来创建枚举并且把它们的整个定义放在一对大括号内：

```
enum enumname {
   // 枚举定义放在这里
}
```

例如我们定义以下表示星期的枚举：

```
import Cocoa

// 定义枚举
enum DaysofaWeek {
    case Sunday
    case Monday
    case TUESDAY
    case WEDNESDAY
    case THURSDAY
    case FRIDAY
    case Saturday
}

var weekDay = DaysofaWeek.THURSDAY
weekDay = .THURSDAY
switch weekDay
{
case .Sunday:
    print("星期天")
case .Monday:
    print("星期一")
case .TUESDAY:
    print("星期二")
case .WEDNESDAY:
    print("星期三")
case .THURSDAY:
    print("星期四")
case .FRIDAY:
    print("星期五")
case .Saturday:
    print("星期六")
}
```

以上程序执行输出结果为：

```
星期四
```

枚举中定义的值（如 `Sunday`，`Monday`，…… 和 `Saturday` ）是这个枚举的成员值（或成员）。`case` 关键词表示一行新的成员值将被定义。

> 注意： 和 C 和 Objective-C 不同，Swift 的枚举成员在被创建时不会被赋予一个默认的整型值。在上面的 `DaysofaWeek` 例子中，`Sunday`，`Monday`，…… 和 `Saturday` 不会隐式地赋值为 `0`，`1`，…… 和 `6` 。相反，这些枚举成员本身就有完备的值，这些值是已经明确定义好的 `DaysofaWeek` 类型。

```
var weekDay = DaysofaWeek.THURSDAY
```

`weekDay` 的类型可以在它被 `DaysofaWeek` 的一个可能值初始化时推断出来。一旦 `weekDay` 被声明为一个 `DaysofaWeek` ，你可以使用一个缩写语法（.）将其设置为另一个 `DaysofaWeek` 的值：

```
var weekDay = .THURSDAY
```

当 `weekDay` 的类型已知时，再次为其赋值可以省略枚举名。使用显式类型的枚举值可以让代码具有更好的可读性。

枚举可分为相关值与原始值。

相关值与原始值的区别

相关值	原始值
不同数据类型	相同数据类型
实例: enum {10,0.8,"Hello"}	实例: enum {10,35,50}
值的创建基于常量或变量	预先填充的值
相关值是当你在创建一个基于枚举成员的新常量或变量时才会被设置，并且每次当你这么做得时候，它的值可以是不同的。	原始值始终是相同的

相关值

以下实例中我们定义一个名为 Student 的枚举类型，它可以是 Name 的一个相关值（Int，Int，Int，Int），或者是 Mark 的一个字符串类型（String）相关值。

```
import Cocoa

enum Student{
    case Name(String)
    case Mark(Int,Int,Int)
}
var studDetails = Student.Name("Runoob")
var studMarks = Student.Mark(98,97,95)
switch studMarks {
case .Name(let studName):
    print("学生的名字是: \(studName)。")
case .Mark(let Mark1, let Mark2, let Mark3):
    print("学生的成绩是: \(Mark1),\(Mark2),\(Mark3)。")
}
```

以上程序执行输出结果为：

```
学生的成绩是: 98,97,95。
```

原始值

原始值可以是字符串，字符，或者任何整型值或浮点型值。每个原始值在它的枚举声明中必须是唯一的。

在原始值为整数的枚举时，不需要显式的为每一个成员赋值，Swift会自动为你赋值。

```
import Cocoa

enum Month: Int {
    case January = 1, February, March, April, May, June, July, Augu
}

let yearMonth = Month.May.rawValue
print("数字月份为: \(yearMonth)。")
```

以上程序执行输出结果为：

```
数字月份为: 5。
```

Swift 结构体

Swift 结构体是构建代码所用的一种通用且灵活的构造体。

我们可以为结构体定义属性（常量、变量）和添加方法，从而扩展结构体的功能。

与 C 和 Objective C 不同的是：

- 结构体不需要包含实现文件和接口。
- 结构体允许我们创建一个单一文件，且系统会自动生成面向其它代码的外部接口。

结构体总是通过被复制的方式在代码中传递，因此它的值是不可修改的。

语法

我们通过关键字 struct 来定义结构体：

```
struct nameStruct {
   Definition 1
   Definition 2
   ……
   Definition N
}
```

实例

我们定义一个名为 MarkStruct 的结构体 ，结构体的属性为学生三个科目的分数，数据类型为 Int：

```
struct MarkStruct{
   var mark1: Int
   var mark2: Int
   var mark3: Int
}
```

我们可以通过结构体名来访问结构体成员。

结构体实例化使用 **let** 关键字：

```
import Cocoa

struct studentMarks {
   var mark1 = 100
   var mark2 = 78
   var mark3 = 98
}
let marks = studentMarks()
print("Mark1 是 \(marks.mark1)")
print("Mark2 是 \(marks.mark2)")
print("Mark3 是 \(marks.mark3)")
```

以上程序执行输出结果为：

```
Mark1 是 100
Mark2 是 78
Mark3 是 98
```

实例中，我们通过结构体名 'studentMarks' 访问学生的成绩。结构体成员初始化为 mark1, mark2, mark3，数据类型为整型。

然后我们通过使用 **let** 关键字将结构体 studentMarks() 实例化并传递给 marks。

最后我们就通过 **.** 号来访问结构体成员的值。

以下实例化通过结构体实例化时传值并克隆一个结构体：

```
import Cocoa

struct MarksStruct {
   var mark: Int

   init(mark: Int) {
      self.mark = mark
   }
}
var aStruct = MarksStruct(mark: 98)
var bStruct = aStruct // aStruct 和 bStruct 是使用相同值的结构体！
bStruct.mark = 97
print(aStruct.mark) // 98
print(bStruct.mark) // 97
```

以上程序执行输出结果为：

```
98
97
```

结构体应用

在你的代码中，你可以使用结构体来定义你的自定义数据类型。

结构体实例总是通过值传递来定义你的自定义数据类型。

按照通用的准则，当符合一条或多条以下条件时，请考虑构建结构体：

- 结构体的主要目的是用来封装少量相关简单数据值。
- 有理由预计一个结构体实例在赋值或传递时，封装的数据将会被拷贝而不是被引用。
- 任何在结构体中储存的值类型属性，也将会被拷贝，而不是被引用。
- 结构体不需要去继承另一个已存在类型的属性或者行为。

举例来说，以下情境中适合使用结构体：

- 几何形状的大小，封装一个 `width` 属性和 `height` 属性，两者均为 `Double` 类型。
- 一定范围内的路径，封装一个 `start` 属性和 `length` 属性，两者均为 `Int` 类型。
- 三维坐标系内一点，封装 `x` ， `y` 和 `z` 属性，三者均为 `Double` 类型。

结构体实例是通过值传递而不是通过引用传递。

```
import Cocoa

struct markStruct{
    var mark1: Int
    var mark2: Int
    var mark3: Int

    init(mark1: Int, mark2: Int, mark3: Int){
        self.mark1 = mark1
        self.mark2 = mark2
        self.mark3 = mark3
    }
}

print("优异成绩:")
var marks = markStruct(mark1: 98, mark2: 96, mark3:100)
print(marks.mark1)
print(marks.mark2)
print(marks.mark3)

print("糟糕成绩:")
var fail = markStruct(mark1: 34, mark2: 42, mark3: 13)
print(fail.mark1)
print(fail.mark2)
print(fail.mark3)
```

以上程序执行输出结果为：

```
优异成绩:
98
96
100
糟糕成绩:
34
42
13
```

以上实例中我们定义了结构体 markStruct，三个成员属性：mark1, mark2 和 mark3。结构体内使用成员属性使用 self 关键字。

从实例中我们可以很好的理解到结构体实例是通过值传递的。

Swift 类

Swift 类是构建代码所用的一种通用且灵活的构造体。

我们可以为类定义属性（常量、变量）和方法。

与其他编程语言所不同的是，Swift 并不要求你为自定义类去创建独立的接口和实现文件。你所要做的是在一个单一文件中定义一个类，系统会自动生成面向其它代码的外部接口。

类和结构体对比

Swift 中类和结构体有很多共同点。共同处在于：

- 定义属性用于存储值
- 定义方法用于提供功能
- 定义附属脚本用于访问值
- 定义构造器用于生成初始化值
- 通过扩展以增加默认实现的功能
- 符合协议以对某类提供标准功能

与结构体相比，类还有如下的附加功能：

- 继承允许一个类继承另一个类的特征
- 类型转换允许在运行时检查和解释一个类实例的类型
- 解构器允许一个类实例释放任何其所被分配的资源
- 引用计数允许对一个类的多次引用

语法:

```
Class classname {
   Definition 1
   Definition 2
   ……
   Definition N
}
```

类定义

```
class student{
   var studname: String
   var mark: Int
   var mark2: Int
}
```

实例化类：

```
let studrecord = student()
```

实例

```
import Cocoa

class MarksStruct {
    var mark: Int
    init(mark: Int) {
        self.mark = mark
    }
}

class studentMarks {
    var mark = 300
}
let marks = studentMarks()
print("成绩为 \(marks.mark)")
```

以上程序执行输出结果为：

```
成绩为 300
```

作为引用类型访问类属性

类的属性可以通过 . 来访问。格式为：实例化类名.属性名：

```
import Cocoa

class MarksStruct {
   var mark: Int
   init(mark: Int) {
      self.mark = mark
   }
}

class studentMarks {
   var mark1 = 300
   var mark2 = 400
   var mark3 = 900
}
let marks = studentMarks()
print("Mark1 is \(marks.mark1)")
print("Mark2 is \(marks.mark2)")
print("Mark3 is \(marks.mark3)")
```

以上程序执行输出结果为：

```
Mark1 is 300
Mark2 is 400
Mark3 is 900
```

恒等运算符

因为类是引用类型，有可能有多个常量和变量在后台同时引用某一个类实例。

为了能够判定两个常量或者变量是否引用同一个类实例，Swift 内建了两个恒等运算符：

恒等运算符	不恒等运算符
运算符为：===	运算符为：!==
如果两个常量或者变量引用同一个类实例则返回 true	如果两个常量或者变量引用不同一个类实例则返回 true

实例

```
import Cocoa

class SampleClass: Equatable {
    let myProperty: String
    init(s: String) {
        myProperty = s
    }
}
func ==(lhs: SampleClass, rhs: SampleClass) -> Bool {
    return lhs.myProperty == rhs.myProperty
}

let spClass1 = SampleClass(s: "Hello")
let spClass2 = SampleClass(s: "Hello")

if spClass1 === spClass2 {// false
    print("引用相同的类实例 \(spClass1)")
}

if spClass1 !== spClass2 {// true
    print("引用不相同的类实例 \(spClass2)")
}
```

以上程序执行输出结果为：

```
引用不相同的类实例  SampleClass
```

Swift 属性

Swift 属性将值跟特定的类、结构或枚举关联。

属性可分为存储属性和计算属性:

存储属性	计算属性
存储常量或变量作为实例的一部分	计算（而不是存储）一个值
用于类和结构体	用于类、结构体和枚举

存储属性和计算属性通常用于特定类型的实例。

属性也可以直接用于类型本身，这种属性称为类型属性。

另外，还可以定义属性观察器来监控属性值的变化，以此来触发一个自定义的操作。属性观察器可以添加到自己写的存储属性上，也可以添加到从父类继承的属性上。

存储属性

简单来说，一个存储属性就是存储在特定类或结构体的实例里的一个常量或变量。

存储属性可以是变量存储属性（用关键字var定义），也可以是常量存储属性（用关键字let定义）。

- 可以在定义存储属性的时候指定默认值
- 也可以在构造过程中设置或修改存储属性的值，甚至修改常量存储属性的值

```
import Cocoa

struct Number
{
   var digits: Int
   let pi = 3.1415
}

var n = Number(digits: 12345)
n.digits = 67

print("\(n.digits)")
print("\(n.pi)")
```

以上程序执行输出结果为：

```
67
3.1415
```

考虑以下代码：

```
let pi = 3.1415
```

代码中 pi 在定义存储属性的时候指定默认值（pi = 3.1415），所以不管你什么时候实例化结构体，它都不会改变。

如果你定义的是一个常量存储属性，如果尝试修改它就会报错，如下所示：

```
import Cocoa

struct Number
{
    var digits: Int
    let numbers = 3.1415
}

var n = Number(digits: 12345)
n.digits = 67

print("\(n.digits)")
print("\(n.numbers)")
n.numbers = 8.7
```

以上程序，执行会报错，错误如下所示：

```
error: cannot assign to property: 'numbers' is a 'let' constant
n.numbers = 8.7
```

意思为 'numbers' 是一个常量，你能修改它。

延迟存储属性

延迟存储属性是指当第一次被调用的时候才会计算其初始值的属性。

在属性声明前使用 **lazy** 来标示一个延迟存储属性。

> 注意：
>
> 必须将延迟存储属性声明成变量（使用 `var` 关键字），因为属性的值在实例构造完成之前可能无法得到。而常量属性在构造过程完成之前必须要有初始值，因此无法声明成延迟属性。

延迟存储属性一般用于：

- 延迟对象的创建。
- 当属性的值依赖于其他未知类

```
import Cocoa

class sample {
    lazy var no = number() // `var` 关键字是必须的
}

class number {
    var name = "Runoob Swift 教程"
}

var firstsample = sample()
print(firstsample.no.name)
```

以上程序执行输出结果为：

```
Runoob Swift 教程
```

实例化变量

如果您有过 Objective-C 经验，应该知道Objective-C 为类实例存储值和引用提供两种方法。对于属性来说，也可以使用实例变量作为属性值的后端存储。

Swift 编程语言中把这些理论统一用属性来实现。Swift 中的属性没有对应的实例变量，属性的后端存储也无法直接访问。这就避免了不同场景下访问方式的困扰，同时也将属性的定义简化成一个语句。

一个类型中属性的全部信息——包括命名、类型和内存管理特征——都在唯一一个地方（类型定义中）定义。

计算属性

除存储属性外，类、结构体和枚举可以定义计算属性，计算属性不直接存储值，而是提供一个 getter 来获取值，一个可选的 setter 来间接设置其他属性或变量的值。

```
import Cocoa

class sample {
    var no1 = 0.0, no2 = 0.0
    var length = 300.0, breadth = 150.0

    var middle: (Double, Double) {
        get{
            return (length / 2, breadth / 2)
        }
        set(axis){
            no1 = axis.0 - (length / 2)
            no2 = axis.1 - (breadth / 2)
        }
    }
}

var result = sample()
print(result.middle)
result.middle = (0.0, 10.0)

print(result.no1)
print(result.no2)
```

以上程序执行输出结果为：

```
(150.0, 75.0)
-150.0
-65.0
```

如果计算属性的 setter 没有定义表示新值的参数名，则可以使用默认名称 newValue。

只读计算属性

只有 getter 没有 setter 的计算属性就是只读计算属性。

只读计算属性总是返回一个值，可以通过点(.)运算符访问，但不能设置新的值。

```
import Cocoa

class film {
    var head = ""
    var duration = 0.0
    var metaInfo: [String:String] {
        return [
            "head": self.head,
            "duration":"\(self.duration)"
        ]
    }
}

var movie = film()
movie.head = "Swift 属性"
movie.duration = 3.09

print(movie.metaInfo["head"]!)
print(movie.metaInfo["duration"]!)
```

以上程序执行输出结果为：

```
Swift 属性
3.09
```

> 注意：
>
> 必须使用 `var` 关键字定义计算属性，包括只读计算属性，因为它们的值不是固定的。 `let` 关键字只用来声明常量属性，表示初始化后再也无法修改的值。

属性观察器

属性观察器监控和响应属性值的变化，每次属性被设置值的时候都会调用属性观察器，甚至新的值和现在的值相同的时候也不例外。

可以为除了延迟存储属性之外的其他存储属性添加属性观察器，也可以通过重载属性的方式为继承的属性（包括存储属性和计算属性）添加属性观察器。

> 注意：
>
> 不需要为无法重载的计算属性添加属性观察器，因为可以通过 setter 直接监控和响应值的变化。

可以为属性添加如下的一个或全部观察器：

- `willSet` 在设置新的值之前调用
- `didSet` 在新的值被设置之后立即调用

- willSet和didSet观察器在属性初始化过程中不会被调用

```
import Cocoa

class Samplepgm {
    var counter: Int = 0{
        willSet(newTotal){
            print("计数器: \(newTotal)")
        }
        didSet{
            if counter > oldValue {
                print("新增数 \(counter - oldValue)")
            }
        }
    }
}
let NewCounter = Samplepgm()
NewCounter.counter = 100
NewCounter.counter = 800
```

以上程序执行输出结果为：

```
计数器: 100
新增数 100
计数器: 800
新增数 700
```

全局变量和局部变量

计算属性和属性观察器所描述的模式也可以用于全局变量和局部变量。

局部变量	全局变量
在函数、方法或闭包内部定义的变量。	函数、方法、闭包或任何类型之外定义的变量。
用于存储和检索值。	用于存储和检索值。
存储属性用于获取和设置值。	存储属性用于获取和设置值。
也用于计算属性。	也用于计算属性。

类型属性

类型属性是作为类型定义的一部分写在类型最外层的花括号（{}）内。

使用关键字 static 来定义值类型的类型属性，关键字 class 来为类定义类型属性。

```
struct Structname {
   static var storedTypeProperty = " "
   static var computedTypeProperty: Int {
      // 这里返回一个 Int 值
   }
}

enum Enumname {
   static var storedTypeProperty = " "
   static var computedTypeProperty: Int {
      // 这里返回一个 Int 值
   }
}

class Classname {
   class var computedTypeProperty: Int {
      // 这里返回一个 Int 值
   }
}
```

> 注意：
>
> 例子中的计算型类型属性是只读的，但也可以定义可读可写的计算型类型属性，跟实例计算属性的语法类似。

获取和设置类型属性的值

类似于实例的属性，类型属性的访问也是通过点运算符(.)来进行。但是，类型属性是通过类型本身来获取和设置，而不是通过实例。实例如下：

```
import Cocoa

struct StudMarks {
   static let markCount = 97
   static var totalCount = 0
   var InternalMarks: Int = 0 {
      didSet {
         if InternalMarks > StudMarks.markCount {
            InternalMarks = StudMarks.markCount
         }
         if InternalMarks > StudMarks.totalCount {
            StudMarks.totalCount = InternalMarks
         }
      }
   }
}

var stud1Mark1 = StudMarks()
var stud1Mark2 = StudMarks()

stud1Mark1.InternalMarks = 98
print(stud1Mark1.InternalMarks)

stud1Mark2.InternalMarks = 87
print(stud1Mark2.InternalMarks)
```

以上程序执行输出结果为：

```
97
87
```

Swift 方法

Swift 方法是与某些特定类型相关联的函数

在 Objective-C 中，类是唯一能定义方法的类型。但在 Swift 中，你不仅能选择是否要定义一个类/结构体/枚举，还能灵活的在你创建的类型（类/结构体/枚举）上定义方法。

实例方法

在 Swift 语言中，实例方法是属于某个特定类、结构体或者枚举类型实例的方法。

实例方法提供以下方法：

- 可以访问和修改实例属性
- 提供与实例目的相关的功能

实例方法要写在它所属的类型的前后大括号({})之间。

实例方法能够隐式访问它所属类型的所有的其他实例方法和属性。

实例方法只能被它所属的类的某个特定实例调用。

实例方法不能脱离于现存的实例而被调用。

语法

```
func funcname(Parameters) -> returntype
{
    Statement1
    Statement2
    ……
    Statement N
    return parameters
}
```

实例

```
import Cocoa

class Counter {
    var count = 0
    func increment() {
        count++
    }
    func incrementBy(amount: Int) {
        count += amount
    }
    func reset() {
        count = 0
    }
}
// 初始计数值是0
let counter = Counter()

// 计数值现在是1
counter.increment()

// 计数值现在是6
counter.incrementBy(5)
print(counter.count)

// 计数值现在是0
counter.reset()
print(counter.count)
```

以上程序执行输出结果为：

```
6
0
```

Counter类定义了三个实例方法：

- `increment` 让计数器按一递增；
- `incrementBy(amount: Int)` 让计数器按一个指定的整数值递增；
- `reset` 将计数器重置为0。

`Counter` 这个类还声明了一个可变属性 `count` ，用它来保持对当前计数器值的追踪。

方法的局部参数名称和外部参数名称

Swift 函数参数可以同时有一个局部名称（在函数体内部使用）和一个外部名称（在调用函数时使用

Swift 中的方法和 Objective-C 中的方法极其相似。像在 Objective-C 中一样，Swift 中方法的名称通常用一个介词指向方法的第一个参数，比如：with，for，by等等。

Swift 默认仅给方法的第一个参数名称一个局部参数名称;默认同时给第二个和后续的参数名称为全局参数名称。

以下实例中 'no1' 在swift中声明为局部参数名称。'no2' 用于全局的声明并通过外部程序访问。

```
import Cocoa

class division {
    var count: Int = 0
    func incrementBy(no1: Int, no2: Int) {
        count = no1 / no2
        print(count)
    }
}

let counter = division()
counter.incrementBy(1800, no2: 3)
counter.incrementBy(1600, no2: 5)
counter.incrementBy(11000, no2: 3)
```

以上程序执行输出结果为：

```
600
320
3666
```

是否提供外部名称设置

我们强制在第一个参数添加外部名称把这个局部名称当作外部名称使用（Swift 2.0 前是使用 # 号）。

```
import Cocoa

class multiplication {
    var count: Int = 0
    func incrementBy(first no1: Int, no2: Int) {
        count = no1 * no2
        print(count)
    }
}

let counter = multiplication()
counter.incrementBy(first: 800, no2: 3)
counter.incrementBy(first: 100, no2: 5)
counter.incrementBy(first: 15000, no2: 3)
```

以上程序执行输出结果为：

```
2400
500
45000
```

self 属性

类型的每一个实例都有一个隐含属性叫做self，self 完全等同于该实例本身。

你可以在一个实例的实例方法中使用这个隐含的self属性来引用当前实例。

```
import Cocoa

class calculations {
    let a: Int
    let b: Int
    let res: Int

    init(a: Int, b: Int) {
        self.a = a
        self.b = b
        res = a + b
        print("Self 内: \(res)")
    }

    func tot(c: Int) -> Int {
        return res - c
    }

    func result() {
        print("结果为: \(tot(20))")
        print("结果为: \(tot(50))")
    }
}

let pri = calculations(a: 600, b: 300)
let sum = calculations(a: 1200, b: 300)

pri.result()
sum.result()
```

以上程序执行输出结果为：

```
Self 内: 900
Self 内: 1500
结果为: 880
结果为: 850
结果为: 1480
结果为: 1450
```

在实例方法中修改值类型

Swift 语言中结构体和枚举是值类型。一般情况下，值类型的属性不能在它的实例方法中被修改。

但是，如果你确实需要在某个具体的方法中修改结构体或者枚举的属性，你可以选择变异(mutating)这个方法，然后方法就可以从方法内部改变它的属性；并且它做的任何改变在方法结束时还会保留在原始结构中。

方法还可以给它隐含的self属性赋值一个全新的实例，这个新实例在方法结束后将替换原来的实例。

```
import Cocoa

struct area {
    var length = 1
    var breadth = 1

    func area() -> Int {
        return length * breadth
    }

    mutating func scaleBy(res: Int) {
        length *= res
        breadth *= res

        print(length)
        print(breadth)
    }
}

var val = area(length: 3, breadth: 5)
val.scaleBy(3)
val.scaleBy(30)
val.scaleBy(300)
```

以上程序执行输出结果为：

```
9
15
270
450
81000
135000
```

在可变方法中给 self 赋值

可变方法能够赋给隐含属性 self 一个全新的实例。

```
import Cocoa

struct area {
    var length = 1
    var breadth = 1

    func area() -> Int {
        return length * breadth
    }

    mutating func scaleBy(res: Int) {
        self.length *= res
        self.breadth *= res
        print(length)
        print(breadth)
    }
}
var val = area(length: 3, breadth: 5)
val.scaleBy(13)
```

以上程序执行输出结果为：

```
39
65
```

类型方法

实例方法是被类型的某个实例调用的方法，你也可以定义类型本身调用的方法，这种方法就叫做类型方法。

声明结构体和枚举的类型方法，在方法的func关键字之前加上关键字static。类可能会用关键字class来允许子类重写父类的实现方法。

类型方法和实例方法一样用点号(.)语法调用。

```
import Cocoa

class Math
{
    class func abs(number: Int) -> Int
    {
        if number < 0
        {
            return (-number)
        }
        else
        {
            return number
        }
    }
}

struct absno
{
    static func abs(number: Int) -> Int
    {
        if number < 0
        {
            return (-number)
        }
        else
        {
            return number
        }
    }
}

let no = Math.abs(-35)
let num = absno.abs(-5)

print(no)
print(num)
```

以上程序执行输出结果为：

```
35
5
```

Swift 下标脚本

下标脚本 可以定义在类（Class）、结构体（structure）和枚举（enumeration）这些目标中，可以认为是访问对象、集合或序列的快捷方式，不需要再调用实例的特定的赋值和访问方法。

举例来说，用下标脚本访问一个数组(Array)实例中的元素可以这样写 someArray[index]，访问字典(Dictionary)实例中的元素可以这样写 someDictionary[key]。

对于同一个目标可以定义多个下标脚本，通过索引值类型的不同来进行重载，而且索引值的个数可以是多个。

下标脚本语法及应用

语法

下标脚本允许你通过在实例后面的方括号中传入一个或者多个的索引值来对实例进行访问和赋值。

语法类似于实例方法和计算型属性的混合。

与定义实例方法类似，定义下标脚本使用subscript关键字，显式声明入参（一个或多个）和返回类型。

与实例方法不同的是下标脚本可以设定为读写或只读。这种方式又有点像计算型属性的getter和setter：

```
subscript(index: Int) -> Int {
    get {
        // 用于下标脚本值的声明
    }
    set(newValue) {
        // 执行赋值操作
    }
}
```

实例 1

```
import Cocoa

struct subexample {
    let decrementer: Int
    subscript(index: Int) -> Int {
        return decrementer / index
    }
}
let division = subexample(decrementer: 100)

print("100 除以 9 等于 \(division[9])")
print("100 除以 2 等于 \(division[2])")
print("100 除以 3 等于 \(division[3])")
print("100 除以 5 等于 \(division[5])")
print("100 除以 7 等于 \(division[7])")
```

以上程序执行输出结果为：

```
100 除以 9 等于 11
100 除以 2 等于 50
100 除以 3 等于 33
100 除以 5 等于 20
100 除以 7 等于 14
```

在上例中，通过 subexample 结构体创建了一个除法运算的实例。数值 100 作为结构体构造函数传入参数初始化实例成员 decrementer。

你可以通过下标脚本来得到结果，比如 division[2] 即为 100 除以 2。

实例 2

```
import Cocoa

class daysofaweek {
    private var days = ["Sunday", "Monday", "Tuesday", "Wednesday",
        "Thursday", "Friday", "saturday"]
    subscript(index: Int) -> String {
        get {
            return days[index]   // 声明下标脚本的值
        }
        set(newValue) {
            self.days[index] = newValue   // 执行赋值操作
        }
    }
}
var p = daysofaweek()

print(p[0])
print(p[1])
print(p[2])
print(p[3])
```

以上程序执行输出结果为：

```
Sunday
Monday
Tuesday
Wednesday
```

用法

根据使用场景不同下标脚本也具有不同的含义。

通常下标脚本是用来访问集合（collection），列表（list）或序列（sequence）中元素的快捷方式。

你可以在你自己特定的类或结构体中自由的实现下标脚本来提供合适的功能。

例如，Swift 的字典（Dictionary）实现了通过下标脚本对其实例中存放的值进行存取操作。在下标脚本中使用和字典索引相同类型的值，并且把一个字典值类型的值赋值给这个下标脚来为字典设值：

```
import Cocoa

var numberOfLegs = ["spider": 8, "ant": 6, "cat": 4]
numberOfLegs["bird"] = 2

print(numberOfLegs)
```

以上程序执行输出结果为：

```
["ant": 6, "bird": 2, "cat": 4, "spider": 8]
```

上例定义一个名为numberOfLegs的变量并用一个字典字面量初始化出了包含三对键值的字典实例。numberOfLegs的字典存放值类型推断为Dictionary<string, int="">。字典实例创建完成之后通过下标脚本的方式将整型值2赋值到字典实例的索引为bird的位置中。</string,>

下标脚本选项

下标脚本允许任意数量的入参索引，并且每个入参类型也没有限制。

下标脚本的返回值也可以是任何类型。

下标脚本可以使用变量参数和可变参数。

一个类或结构体可以根据自身需要提供多个下标脚本实现，在定义下标脚本时通过传入参数的类型进行区分，使用下标脚本时会自动匹配合适的下标脚本实现运行，这就是下标脚本的重载。

```
import Cocoa

struct Matrix {
    let rows: Int, columns: Int
    var print: [Double]
    init(rows: Int, columns: Int) {
        self.rows = rows
        self.columns = columns
        print = Array(count: rows * columns, repeatedValue: 0.0)
    }
    subscript(row: Int, column: Int) -> Double {
        get {
            return print[(row * columns) + column]
        }
        set {
            print[(row * columns) + column] = newValue
        }
    }
}
// 创建了一个新的 3 行 3 列的Matrix实例
var mat = Matrix(rows: 3, columns: 3)

// 通过下标脚本设置值
mat[0,0] = 1.0
mat[0,1] = 2.0
mat[1,0] = 3.0
mat[1,1] = 5.0

// 通过下标脚本获取值
print("\(mat[0,0])")
print("\(mat[0,1])")
print("\(mat[1,0])")
print("\(mat[1,1])")
```

以上程序执行输出结果为：

```
1.0
2.0
3.0
5.0
```

Matrix 结构体提供了一个两个传入参数的构造方法，两个参数分别是rows和columns，创建了一个足够容纳rows * columns个数的Double类型数组。为了存储，将数组的大小和数组每个元素初始值0.0。

你可以通过传入合适的row和column的数量来构造一个新的Matrix实例。

Swift 继承

继承我们可以理解为一个类获取了另外一个类的方法和属性。

当一个类继承其它类时，继承类叫子类，被继承类叫超类（或父类）

在 Swift 中，类可以调用和访问超类的方法，属性和下标脚本，并且可以重写它们。

我们也可以为类中继承来的属性添加属性观察器。

基类

没有继承其它类的类，称之为基类（Base Class）。

以下实例中我们定义了基类 StudDetails ，描述了学生（stname）及其各科成绩的分数(mark1、mark2、mark3)：

```
class StudDetails {
    var stname: String!
    var mark1: Int!
    var mark2: Int!
    var mark3: Int!
    init(stname: String, mark1: Int, mark2: Int, mark3: Int) {
        self.stname = stname
        self.mark1 = mark1
        self.mark2 = mark2
        self.mark3 = mark3
    }
}
let stname = "swift"
let mark1 = 98
let mark2 = 89
let mark3 = 76

print(stname)
print(mark1)
print(mark2)
print(mark3)
```

以上程序执行输出结果为：

```
swift
98
89
76
```

```
swift
98
89
76
```

子类

子类指的是在一个已有类的基础上创建一个新的类。

为了指明某个类的超类，将超类名写在子类名的后面，用冒号(:)分隔,语法格式如下

```
class SomeClass: SomeSuperclass {
    // 类的定义
}
```

实例

以下实例中我们定义了超类 StudDetails，然后使用子类 Tom 继承它：

```
class StudDetails
{
    var mark1: Int;
    var mark2: Int;

    init(stm1:Int, results stm2:Int)
    {
        mark1 = stm1;
        mark2 = stm2;
    }

    func show()
    {
        print("Mark1:\(self.mark1), Mark2:\(self.mark2)")
    }
}

class Tom : StudDetails
{
    init()
    {
        super.init(stm1: 93, results: 89)
    }
}

let tom = Tom()
tom.show()
```

以上程序执行输出结果为：

```
Mark1:93,  Mark2:89
```

重写（Overriding）

子类可以通过继承来的实例方法，类方法，实例属性，或下标脚本来实现自己的定制功能，我们把这种行为叫重写（overriding）。

我们可以使用 override 关键字来实现重写。

访问超类的方法、属性及下标脚本

你可以通过使用super前缀来访问超类的方法，属性或下标脚本。

重写	访问方法，属性，下标脚本
方法	super.somemethod()
属性	super.someProperty()
下标脚本	super[someIndex]

重写方法和属性

重写方法

在我们的子类中我们可以使用 override 关键字来重写超类的方法。

以下实例中我们重写了 show() 方法：

```
class SuperClass {
    func show() {
        print("这是超类 SuperClass")
    }
}

class SubClass: SuperClass  {
    override func show() {
        print("这是子类 SubClass")
    }
}

let superClass = SuperClass()
superClass.show()

let subClass = SubClass()
subClass.show()
```

以上程序执行输出结果为：

```
这是超类 SuperClass
这是子类 SubClass
```

重写属性

你可以提供定制的 getter（或 setter）来重写任意继承来的属性，无论继承来的属性是存储型的还是计算型的属性。

子类并不知道继承来的属性是存储型的还是计算型的，它只知道继承来的属性会有一个名字和类型。所以你在重写一个属性时，必需将它的名字和类型都写出来。

注意点：

- 如果你在重写属性中提供了 setter，那么你也一定要提供 getter。
- 如果你不想在重写版本中的 getter 里修改继承来的属性值，你可以直接通过super.someProperty来返回继承来的值，其中someProperty是你要重写的属性的名字。

以下实例我们定义了超类 Circle 及子类 Rectangle, 在 Rectangle 类中我们重写属性 area：

```
class Circle {
    var radius = 12.5
    var area: String {
        return "矩形半径 \(radius) "
    }
}

// 继承超类 Circle
class Rectangle: Circle {
    var print = 7
    override var area: String {
        return super.area + " ，但现在被重写为 \(print)"
    }
}

let rect = Rectangle()
rect.radius = 25.0
rect.print = 3
print("Radius \(rect.area)")
```

以上程序执行输出结果为：

```
Radius  矩形半径  25.0  ，但现在被重写为  3
```

重写属性观察器

你可以在属性重写中为一个继承来的属性添加属性观察器。这样一来，当继承来的属性值发生改变时，你就会监测到。

注意：你不可以为继承来的常量存储型属性或继承来的只读计算型属性添加属性观察器。

```
class Circle {
    var radius = 12.5
    var area: String {
        return "矩形半径为 \(radius) "
    }
}

class Rectangle: Circle {
    var print = 7
    override var area: String {
        return super.area + " ，但现在被重写为 \(print)"
    }
}


let rect = Rectangle()
rect.radius = 25.0
rect.print = 3
print("半径: \(rect.area)")

class Square: Rectangle {
    override var radius: Double {
        didSet {
            print = Int(radius/5.0)+1
        }
    }
}


let sq = Square()
sq.radius = 100.0
print("半径: \(sq.area)")
```

```
半径:  矩形半径为  25.0  ，但现在被重写为  3
半径:  矩形半径为  100.0  ，但现在被重写为  21
```

防止重写

我们可以使用 final 关键字防止它们被重写。

如果你重写了final方法，属性或下标脚本，在编译时会报错。

你可以通过在关键字class前添加final特性（final class）来将整个类标记为 final 的，这样的类是不可被继承的，否则会报编译错误。

```
final class Circle {
    final var radius = 12.5
    var area: String {
        return "矩形半径为 \(radius) "
    }
}
class Rectangle: Circle {
    var print = 7
    override var area: String {
        return super.area + " ，但现在被重写为 \(print)"
    }
}

let rect = Rectangle()
rect.radius = 25.0
rect.print = 3
print("半径: \(rect.area)")

class Square: Rectangle {
    override var radius: Double {
        didSet {
            print = Int(radius/5.0)+1
        }
    }
}

let sq = Square()
sq.radius = 100.0
print("半径: \(sq.area)")
```

由于以上实例使用了 final 关键字不允许重写，所以执行会报错：

```
error: var overrides a 'final' var
    override var area: String {
                 ^
note: overridden declaration is here
    var area: String {
        ^
error: var overrides a 'final' var
    override var radius: Double {
                 ^
note: overridden declaration is here
    final var radius = 12.5
              ^
error: inheritance from a final class 'Circle'
class Rectangle: Circle {
      ^
```

Swift 构造过程

构造过程是为了使用某个类、结构体或枚举类型的实例而进行的准备过程。这个过程包含了为实例中的每个属性设置初始值和为其执行必要的准备和初始化任务。

Swift 构造函数使用 init() 方法。

与 Objective-C 中的构造器不同，Swift 的构造器无需返回值，它们的主要任务是保证新实例在第一次使用前完成正确的初始化。

类实例也可以通过定义析构器（deinitializer）在类实例释放之前执行清理内存的工作。

存储型属性的初始赋值

类和结构体在实例创建时，必须为所有存储型属性设置合适的初始值。

存储属性在构造器中赋值时，它们的值是被直接设置的，不会触发任何属性观测器。

存储属性在构造器中赋值流程：

- 创建初始值。
- 在属性定义中指定默认属性值。
- 初始化实例，并调用 init() 方法。

构造器

构造器在创建某特定类型的新实例时调用。它的最简形式类似于一个不带任何参数的实例方法，以关键字init命名。

语法

```
init()
{
    // 实例化后执行的代码
}
```

实例

以下结构体定义了一个不带参数的构造器 init，并在里面将存储型属性 length 和 breadth 的值初始化为 6 和 12：

```
struct rectangle {
    var length: Double
    var breadth: Double
    init() {
        length = 6
        breadth = 12
    }
}
var area = rectangle()
print("矩形面积为 \(area.length*area.breadth)")
```

以上程序执行输出结果为：

```
矩形面积为 72.0
```

默认属性值

我们可以在构造器中为存储型属性设置初始值；同样，也可以在属性声明时为其设置默认值。

> 使用默认值能让你的构造器更简洁、更清晰，且能通过默认值自动推导出属性的类型。

以下实例我们在属性声明时为其设置默认值：

```
struct rectangle {
    // 设置默认值
    var length = 6
    var breadth = 12
}
var area = rectangle()
print("矩形的面积为 \(area.length*area.breadth)")
```

以上程序执行输出结果为：

```
矩形面积为 72
```

构造参数

你可以在定义构造器 init() 时提供构造参数，如下所示：

```
struct Rectangle {
    var length: Double
    var breadth: Double
    var area: Double

    init(fromLength length: Double, fromBreadth breadth: Double) {
        self.length = length
        self.breadth = breadth
        area = length * breadth
    }

    init(fromLeng leng: Double, fromBread bread: Double) {
        self.length = leng
        self.breadth = bread
        area = leng * bread
    }
}

let ar = Rectangle(fromLength: 6, fromBreadth: 12)
print("面积为: \(ar.area)")

let are = Rectangle(fromLeng: 36, fromBread: 12)
print("面积为: \(are.area)")
```

以上程序执行输出结果为：

```
面积为:  72.0
面积为:  432.0
```

内部和外部参数名

跟函数和方法参数相同，构造参数也存在一个在构造器内部使用的参数名字和一个在调用构造器时使用的外部参数名字。

然而，构造器并不像函数和方法那样在括号前有一个可辨别的名字。所以在调用构造器时，主要通过构造器中的参数名和类型来确定需要调用的构造器。

如果你在定义构造器时没有提供参数的外部名字，Swift 会为每个构造器的参数自动生成一个跟内部名字相同的外部名。

```
struct Color {
    let red, green, blue: Double
    init(red: Double, green: Double, blue: Double) {
        self.red   = red
        self.green = green
        self.blue  = blue
    }
    init(white: Double) {
        red   = white
        green = white
        blue  = white
    }
}

// 创建一个新的Color实例，通过三种颜色的外部参数名来传值，并调用构造器
let magenta = Color(red: 1.0, green: 0.0, blue: 1.0)

print("red 值为: \(magenta.red)")
print("green 值为: \(magenta.green)")
print("blue 值为: \(magenta.blue)")

// 创建一个新的Color实例，通过三种颜色的外部参数名来传值，并调用构造器
let halfGray = Color(white: 0.5)
print("red 值为: \(halfGray.red)")
print("green 值为: \(halfGray.green)")
print("blue 值为: \(halfGray.blue)")
```

以上程序执行输出结果为：

```
red 值为: 1.0
green 值为: 0.0
blue 值为: 1.0
red 值为: 0.5
green 值为: 0.5
blue 值为: 0.5
```

没有外部名称参数

如果你不希望为构造器的某个参数提供外部名字，你可以使用下划线 `_` 来显示描述它的外部名。

```
struct Rectangle {
    var length: Double

    init(frombreadth breadth: Double) {
        length = breadth * 10
    }

    init(frombre bre: Double) {
        length = bre * 30
    }
    //不提供外部名字
    init(_ area: Double) {
        length = area
    }
}

// 调用不提供外部名字
let rectarea = Rectangle(180.0)
print("面积为: \(rectarea.length)")

// 调用不提供外部名字
let rearea = Rectangle(370.0)
print("面积为: \(rearea.length)")

// 调用不提供外部名字
let recarea = Rectangle(110.0)
print("面积为: \(recarea.length)")
```

以上程序执行输出结果为：

```
面积为: 180.0
面积为: 370.0
面积为: 110.0
```

可选属性类型

如果你定制的类型包含一个逻辑上允许取值为空的存储型属性，你都需要将它定义为可选类型optional type（可选属性类型）。

当存储属性声明为可选时，将自动初始化为空 nil。

```
struct Rectangle {
    var length: Double?

    init(frombreadth breadth: Double) {
        length = breadth * 10
    }

    init(frombre bre: Double) {
        length = bre * 30
    }

    init(_ area: Double) {
        length = area
    }
}

let rectarea = Rectangle(180.0)
print("面积为：\(rectarea.length)")

let rearea = Rectangle(370.0)
print("面积为：\(rearea.length)")

let recarea = Rectangle(110.0)
print("面积为：\(recarea.length)")
```

以上程序执行输出结果为：

```
面积为：Optional(180.0)
面积为：Optional(370.0)
面积为：Optional(110.0)
```

构造过程中修改常量属性

只要在构造过程结束前常量的值能确定，你可以在构造过程中的任意时间点修改常量属性的值。

对某个类实例来说，它的常量属性只能在定义它的类的构造过程中修改；不能在子类中修改。

尽管 length 属性现在是常量，我们仍然可以在其类的构造器中设置它的值：

```
struct Rectangle {
    let length: Double?

    init(frombreadth breadth: Double) {
        length = breadth * 10
    }

    init(frombre bre: Double) {
        length = bre * 30
    }

    init(_ area: Double) {
        length = area
    }
}

let rectarea = Rectangle(180.0)
print("面积为：\(rectarea.length)")

let rearea = Rectangle(370.0)
print("面积为：\(rearea.length)")

let recarea = Rectangle(110.0)
print("面积为：\(recarea.length)")
```

以上程序执行输出结果为：

```
面积为：Optional(180.0)
面积为：Optional(370.0)
面积为：Optional(110.0)
```

默认构造器

默认构造器将简单的创建一个所有属性值都设置为默认值的实例:

以下实例中，ShoppingListItem类中的所有属性都有默认值，且它是没有父类的基类，它将自动获得一个可以为所有属性设置默认值的默认构造器

```
class ShoppingListItem {
    var name: String?
    var quantity = 1
    var purchased = false
}
var item = ShoppingListItem()


print("名字为: \(item.name)")
print("数理为: \(item.quantity)")
print("是否付款: \(item.purchased)")
```

以上程序执行输出结果为：

```
名字为:  nil
数理为:  1
是否付款:  false
```

结构体的逐一成员构造器

如果结构体对所有存储型属性提供了默认值且自身没有提供定制的构造器，它们能自动获得一个逐一成员构造器。

我们在调用逐一成员构造器时，通过与成员属性名相同的参数名进行传值来完成对成员属性的初始赋值。

下面例子中定义了一个结构体 Rectangle，它包含两个属性 length 和 breadth。Swift 可以根据这两个属性的初始赋值100.0 、200.0自动推导出它们的类型Double。

```
struct Rectangle {
    var length = 100.0, breadth = 200.0
}
let area = Rectangle(length: 24.0, breadth: 32.0)

print("矩形的面积: \(area.length)")
print("矩形的面积: \(area.breadth)")
```

由于这两个存储型属性都有默认值，结构体 Rectangle 自动获得了一个逐一成员构造器 init(width:height:)。 你可以用它来为 Rectangle 创建新的实例。

以上程序执行输出结果为：

```
名字为:  nil
矩形的面积:  24.0
矩形的面积:  32.0
```

值类型的构造器代理

构造器可以通过调用其它构造器来完成实例的部分构造过程。这一过程称为构造器代理，它能减少多个构造器间的代码重复。

以下实例中，Rect 结构体调用了 Size 和 Point 的构造过程：

```
struct Size {
    var width = 0.0, height = 0.0
}
struct Point {
    var x = 0.0, y = 0.0
}

struct Rect {
    var origin = Point()
    var size = Size()
    init() {}
    init(origin: Point, size: Size) {
        self.origin = origin
        self.size = size
    }
    init(center: Point, size: Size) {
        let originX = center.x - (size.width / 2)
        let originY = center.y - (size.height / 2)
        self.init(origin: Point(x: originX, y: originY), size: size
    }
}


// origin和size属性都使用定义时的默认值Point(x: 0.0, y: 0.0)和Size(widt
let basicRect = Rect()
print("Size 结构体初始值: \(basicRect.size.width, basicRect.size.heig
print("Rect 结构体初始值: \(basicRect.origin.x, basicRect.origin.y) '

// 将origin和size的参数值赋给对应的存储型属性
let originRect = Rect(origin: Point(x: 2.0, y: 2.0),
    size: Size(width: 5.0, height: 5.0))

print("Size 结构体初始值: \(originRect.size.width, originRect.size.he
print("Rect 结构体初始值: \(originRect.origin.x, originRect.origin.y


//先通过center和size的值计算出origin的坐标。
//然后再调用（或代理给）init(origin:size:)构造器来将新的origin和size值赋值
let centerRect = Rect(center: Point(x: 4.0, y: 4.0),
    size: Size(width: 3.0, height: 3.0))

print("Size 结构体初始值: \(centerRect.size.width, centerRect.size.he
print("Rect 结构体初始值: \(centerRect.origin.x, centerRect.origin.y
```

以上程序执行输出结果为：

```
Size 结构体初始值: (0.0, 0.0)
Rect 结构体初始值: (0.0, 0.0)
Size 结构体初始值: (5.0, 5.0)
Rect 结构体初始值: (2.0, 2.0)
Size 结构体初始值: (3.0, 3.0)
Rect 结构体初始值: (2.5, 2.5)
```

构造器代理规则

值类型	类类型
不支持继承，所以构造器代理的过程相对简单，因为它们只能代理给本身提供的其它构造器。 你可以使用self.init在自定义的构造器中引用其它的属于相同值类型的构造器。	它可以继承自其它类,这意味着类有责任保证其所有继承的存储型属性在构造时也能正确的初始化。

类的继承和构造过程

Swift 提供了两种类型的类构造器来确保所有类实例中存储型属性都能获得初始值，它们分别是指定构造器和便利构造器。

指定构造器	便利构造器
类中最主要的构造器	类中比较次要的、辅助型的构造器
初始化类中提供的所有属性，并根据父类链往上调用父类的构造器来实现父类的初始化。	可以定义便利构造器来调用同一个类中并为其参数提供默认值。你也可以定义建一个特殊用途或特定输入的实例。
每一个类都必须拥有至少一个指定构造器	只在必要的时候为类提供便利构造器
`Init(parameters) { statements }`	`convenience init(parameters) {`

指定构造器实例

```
class mainClass {
    var no1 : Int // 局部存储变量
    init(no1 : Int) {
        self.no1 = no1 // 初始化
    }
}
class subClass : mainClass {
    var no2 : Int // 新的子类存储变量
    init(no1 : Int, no2 : Int) {
        self.no2 = no2 // 初始化
        super.init(no1:no1) // 初始化超类
    }
}

let res = mainClass(no1: 10)
let res2 = subClass(no1: 10, no2: 20)

print("res 为: \(res.no1)")
print("res2 为: \(res2.no1)")
print("res2 为: \(res2.no2)")
```

以上程序执行输出结果为：

```
res 为: 10
res 为: 10
res 为: 20
```

便利构造器实例

```
class mainClass {
    var no1 : Int // 局部存储变量
    init(no1 : Int) {
        self.no1 = no1 // 初始化
    }
}

class subClass : mainClass {
    var no2 : Int
    init(no1 : Int, no2 : Int) {
        self.no2 = no2
        super.init(no1:no1)
    }
    // 便利方法只需要一个参数
    override convenience init(no1: Int)  {
        self.init(no1:no1, no2:0)
    }
}
let res = mainClass(no1: 20)
let res2 = subClass(no1: 30, no2: 50)

print("res 为: \(res.no1)")
print("res2 为: \(res2.no1)")
print("res2 为: \(res2.no2)")
```

以上程序执行输出结果为：

```
res 为:  20
res2 为:  30
res2 为:  50
```

构造器的继承和重载

Swift 中的子类不会默认继承父类的构造器。

父类的构造器仅在确定和安全的情况下被继承。

当你重写一个父类指定构造器时，你需要写override修饰符。

```
class SuperClass {
    var corners = 4
    var description: String {
        return "\(corners) 边"
    }
}
let rectangle = SuperClass()
print("矩形: \(rectangle.description)")

class SubClass: SuperClass {
    override init() {  //重载构造器
        super.init()
        corners = 5
    }
}

let subClass = SubClass()
print("五角型: \(subClass.description)")
```

以上程序执行输出结果为：

```
矩形:  4  边
五角型:  5  边
```

指定构造器和便利构造器实例

接下来的例子将在操作中展示指定构造器、便利构造器和自动构造器的继承。

它定义了包含两个个类MainClass、SubClass的类层次结构，并将演示它们的构造器是如何相互作用的。

```
class MainClass {
    var name: String

    init(name: String) {
        self.name = name
    }

    convenience init() {
        self.init(name: "[匿名]")
    }
}
let main = MainClass(name: "Runoob")
print("MainClass 名字为: \(main.name)")

let main2 = MainClass()
print("没有对应名字: \(main2.name)")

class SubClass: MainClass {
    var count: Int
    init(name: String, count: Int) {
        self.count = count
        super.init(name: name)
    }

    override convenience init(name: String) {
        self.init(name: name, count: 1)
    }
}

let sub = SubClass(name: "Runoob")
print("MainClass 名字为: \(sub.name)")

let sub2 = SubClass(name: "Runoob", count: 3)
print("count 变量: \(sub2.count)")
```

以上程序执行输出结果为：

```
MainClass 名字为: Runoob
没有对应名字: [匿名]
MainClass 名字为: Runoob
count 变量: 3
```

类的可失败构造器

如果一个类，结构体或枚举类型的对象，在构造自身的过程中有可能失败，则为其定义一个可失败构造器。

变量初始化失败可能的原因有：

- 传入无效的参数值。
- 缺少某种所需的外部资源。
- 没有满足特定条件。

为了妥善处理这种构造过程中可能会失败的情况。

你可以在一个类，结构体或是枚举类型的定义中，添加一个或多个可失败构造器。其语法为在init关键字后面加添问号(init?)。

实例

下例中，定义了一个名为Animal的结构体，其中有一个名为species的，String类型的常量属性。

同时该结构体还定义了一个，带一个String类型参数species的,可失败构造器。这个可失败构造器，被用来检查传入的参数是否为一个空字符串，如果为空字符串，则该可失败构造器，构建对象失败，否则成功。

```
struct Animal {
    let species: String
    init?(species: String) {
        if species.isEmpty { return nil }
        self.species = species
    }
}

//通过该可失败构造器来构建一个Animal的对象，并检查其构建过程是否成功
// someCreature 的类型是 Animal? 而不是 Animal
let someCreature = Animal(species: "长颈鹿")

// 打印 "动物初始化为长颈鹿"
if let giraffe = someCreature {
    print("动物初始化为\(giraffe.species)")
}
```

以上程序执行输出结果为：

```
动物初始化为长颈鹿
```

枚举类型的可失败构造器

你可以通过构造一个带一个或多个参数的可失败构造器来获取枚举类型中特定的枚举成员。

实例

下例中，定义了一个名为TemperatureUnit的枚举类型。其中包含了三个可能的枚举成员(Kelvin，Celsius，和 Fahrenheit)和一个被用来找到Character值所对应的枚举成员的可失败构造器：

```
enum TemperatureUnit {
    // 开尔文，摄氏，华氏
    case Kelvin, Celsius, Fahrenheit
    init?(symbol: Character) {
        switch symbol {
        case "K":
            self = .Kelvin
        case "C":
            self = .Celsius
        case "F":
            self = .Fahrenheit
        default:
            return nil
        }
    }
}


let fahrenheitUnit = TemperatureUnit(symbol: "F")
if fahrenheitUnit != nil {
    print("这是一个已定义的温度单位，所以初始化成功。")
}

let unknownUnit = TemperatureUnit(symbol: "X")
if unknownUnit == nil {
    print("这不是一个已定义的温度单位，所以初始化失败。")
}
```

以上程序执行输出结果为：

```
这是一个已定义的温度单位，所以初始化成功。
这不是一个已定义的温度单位，所以初始化失败。
```

类的可失败构造器

值类型（如结构体或枚举类型）的可失败构造器，对何时何地触发构造失败这个行为没有任何的限制。

但是，类的可失败构造器只能在所有的类属性被初始化后和所有类之间的构造器之间的代理调用发生完后触发失败行为。

实例

下例子中，定义了一个名为 StudRecord 的类，因为 studname 属性是一个常量，所以一旦 StudRecord 类构造成功，studname 属性肯定有一个非nil的值。

```
class StudRecord {
    let studname: String!
    init?(studname: String) {
        self.studname = studname
        if studname.isEmpty { return nil }
    }
}
if let stname = StudRecord(studname: "失败构造器") {
    print("模块为 \(stname.studname)")
}
```

以上程序执行输出结果为：

```
模块为 失败构造器
```

覆盖一个可失败构造器

就如同其它构造器一样，你也可以用子类的可失败构造器覆盖基类的可失败构造器。

者你也可以用子类的非可失败构造器覆盖一个基类的可失败构造器。

你可以用一个非可失败构造器覆盖一个可失败构造器，但反过来却行不通。

一个非可失败的构造器永远也不能代理调用一个可失败构造器。

实例

以下实例描述了可失败与非可失败构造器：

```
class Planet {
    var name: String

    init(name: String) {
        self.name = name
    }

    convenience init() {
        self.init(name: "[No Planets]")
    }
}
let plName = Planet(name: "Mercury")
print("行星的名字是: \(plName.name)")

let noplName = Planet()
print("没有这个名字的行星: \(noplName.name)")

class planets: Planet {
    var count: Int

    init(name: String, count: Int) {
        self.count = count
        super.init(name: name)
    }

    override convenience init(name: String) {
        self.init(name: name, count: 1)
    }
}
```

以上程序执行输出结果为：

```
行星的名字是: Mercury
没有这个名字的行星: [No Planets]
```

可失败构造器 init!

通常来说我们通过在init关键字后添加问号的方式（init?）来定义一个可失败构造器，但你也可以使用通过在init后面添加惊叹号的方式来定义一个可失败构造器(init!)。实例如下：

```
struct StudRecord {
    let stname: String

    init!(stname: String) {
        if stname.isEmpty {return nil }
        self.stname = stname
    }
}

let stmark = StudRecord(stname: "Runoob")
if let name = stmark {
    print("指定了学生名")
}

let blankname = StudRecord(stname: "")
if blankname == nil {
    print("学生名为空")
}
```

以上程序执行输出结果为：

```
指定了学生名  学生名为空
```

Swift 析构过程

在一个类的实例被释放之前，析构函数被立即调用。用关键字 `deinit` 来标示析构函数，类似于初始化函数用 `init` 来标示。析构函数只适用于类类型。

析构过程原理

Swift 会自动释放不再需要的实例以释放资源。

Swift 通过自动引用计数（ARC）处理实例的内存管理。

通常当你的实例被释放时不需要手动地去清理。但是，当使用自己的资源时，你可能需要进行一些额外的清理。

例如，如果创建了一个自定义的类来打开一个文件，并写入一些数据，你可能需要在类实例被释放之前关闭该文件。

语法

在类的定义中，每个类最多只能有一个析构函数。析构函数不带任何参数，在写法上不带括号：

```
deinit {
    // 执行析构过程
}
```

实例

```
var counter = 0;  // 引用计数器
class BaseClass {
    init() {
        counter++;
    }
    deinit {
        counter--;
    }
}

var show: BaseClass? = BaseClass()
print(counter)
show = nil
print(counter)
```

以上程序执行输出结果为：

```
1
0
```

当 show = nil 语句执行后，计算器减去 1，show 占用的内存就会释放。

```
var counter = 0;  // 引用计数器

class BaseClass {
    init() {
        counter++;
    }

    deinit {
        counter--;
    }
}

var show: BaseClass? = BaseClass()

print(counter)
print(counter)
```

以上程序执行输出结果为：

```
1
1
```

Swift 可选链

可选链（Optional Chaining）是一种是一种可以请求和调用属性、方法和子脚本的过程，用于请求或调用的目标可能为nil。

可选链返回两个值：

- 如果目标有值，调用就会成功，返回该值
- 如果目标为nil，调用将返回nil

多次请求或调用可以被链接成一个链，如果任意一个节点为nil将导致整条链失效。

可选链可替代强制解析

通过在属性、方法、或下标脚本的可选值后面放一个问号(?)，即可定义一个可选链。

可选链 **'?'**	感叹号（**!**）**强**制展开方法，属性，下标脚本可选链
? 放置于可选值后来调用方法，属性，下标脚本	! 放置于可选值后来调用方法，属性，下标脚本来强制展开值
当可选为 nil 输出比较友好的错误信息	当可选为 nil 时强制展开执行错误

使用感叹号(!)可选链实例

```
class Person {
    var residence: Residence?
}

class Residence {
    var numberOfRooms = 1
}

let john = Person()

//将导致运行时错误
let roomCount = john.residence!.numberOfRooms
```

以上程序执行输出结果为：

```
fatal error: unexpectedly found nil while unwrapping an Optional va
```

使用感叹号(!)可选链实例

```
class Person {
    var residence: Residence?
}

class Residence {
    var numberOfRooms = 1
}

let john = Person()

// 链接可选residence?属性，如果residence存在则取回numberOfRooms的值
if let roomCount = john.residence?.numberOfRooms {
    print("John 的房间号为 \(roomCount)。")
} else {
    print("不能查看房间号")
}
```

以上程序执行输出结果为：

```
不能查看房间号
```

因为这种尝试获得numberOfRooms的操作有可能失败，可选链会返回Int?类型值，或者称作"可选Int"。当residence是空的时候（上例），选择Int将会为空，因此会出现无法访问numberOfRooms的情况。

要注意的是，即使numberOfRooms是非可选Int（Int?）时这一点也成立。只要是通过可选链的请求就意味着最后numberOfRooms总是返回一个Int?而不是Int。

为可选链定义模型类

你可以使用可选链来多层调用属性，方法，和下标脚本。这让你可以利用它们之间的复杂模型来获取更底层的属性，并检查是否可以成功获取此类底层属性。

实例

定义了四个模型类，其中包括多层可选链：

```
class Person {
    var residence: Residence?
}

// 定义了一个变量 rooms，它被初始化为一个Room[]类型的空数组
class Residence {
    var rooms = [Room]()
    var numberOfRooms: Int {
        return rooms.count
    }
    subscript(i: Int) -> Room {
        return rooms[i]
    }
    func printNumberOfRooms() {
        print("房间号为 \(numberOfRooms)")
    }
    var address: Address?
}

// Room 定义一个name属性和一个设定room名的初始化器
class Room {
    let name: String
    init(name: String) { self.name = name }
}

// 模型中的最终类叫做Address
class Address {
    var buildingName: String?
    var buildingNumber: String?
    var street: String?
    func buildingIdentifier() -> String? {
        if (buildingName != nil) {
            return buildingName
        } else if (buildingNumber != nil) {
            return buildingNumber
        } else {
            return nil
        }
    }
}
```

通过可选链调用方法

你可以使用可选链的来调用可选值的方法并检查方法调用是否成功。即使这个方法没有返回值，你依然可以使用可选链来达成这一目的。

```
class Person {
    var residence: Residence?
}

// 定义了一个变量 rooms，它被初始化为一个Room[]类型的空数组
class Residence {
    var rooms = [Room]()
    var numberOfRooms: Int {
        return rooms.count
    }
    subscript(i: Int) -> Room {
        return rooms[i]
    }
    func printNumberOfRooms() {
        print("房间号为 \(numberOfRooms)")
    }
    var address: Address?
}

// Room 定义一个name属性和一个设定room名的初始化器
class Room {
    let name: String
    init(name: String) { self.name = name }
}

// 模型中的最终类叫做Address
class Address {
    var buildingName: String?
    var buildingNumber: String?
    var street: String?
    func buildingIdentifier() -> String? {
        if (buildingName != nil) {
            return buildingName
        } else if (buildingNumber != nil) {
            return buildingNumber
        } else {
            return nil
        }
    }
}

let john = Person()

if ((john.residence?.printNumberOfRooms()) != nil) {
    print("输出房间号")
} else {
    print("无法输出房间号")
}
```

以上程序执行输出结果为：

```
无法输出房间号
```

使用if语句来检查是否能成功调用printNumberOfRooms方法：如果方法通过可选链调用成功，printNumberOfRooms的隐式返回值将会是Void，如果没有成功，将返回nil。

使用可选链调用下标脚本

你可以使用可选链来尝试从下标脚本获取值并检查下标脚本的调用是否成功，然而，你不能通过可选链来设置下标脚本。

实例1

```
class Person {
    var residence: Residence?
}

// 定义了一个变量 rooms，它被初始化为一个Room[]类型的空数组
class Residence {
    var rooms = [Room]()
    var numberOfRooms: Int {
        return rooms.count
    }
    subscript(i: Int) -> Room {
        return rooms[i]
    }
    func printNumberOfRooms() {
        print("房间号为 \(numberOfRooms)")
    }
    var address: Address?
}

// Room 定义一个name属性和一个设定room名的初始化器
class Room {
    let name: String
    init(name: String) { self.name = name }
}

// 模型中的最终类叫做Address
class Address {
    var buildingName: String?
    var buildingNumber: String?
    var street: String?
    func buildingIdentifier() -> String? {
        if (buildingName != nil) {
            return buildingName
        } else if (buildingNumber != nil) {
            return buildingNumber
        } else {
            return nil
        }
    }
}

let john = Person()
if let firstRoomName = john.residence?[0].name {
    print("第一个房间名 \(firstRoomName).")
} else {
    print("无法检索到房间")
}
```

以上程序执行输出结果为：

```
无法检索到房间
```

在下标脚本调用中可选链的问号直接跟在 circname.print 的后面，在下标脚本括号的前面，因为circname.print是可选链试图获得的可选值。

实例2

实例中创建一个Residence实例给john.residence，且在他的rooms数组中有一个或多个Room实例，那么你可以使用可选链通过Residence下标脚本来获取在rooms数组中的实例了：

```
class Person {
    var residence: Residence?
}

// 定义了一个变量 rooms，它被初始化为一个Room[]类型的空数组
class Residence {
    var rooms = [Room]()
    var numberOfRooms: Int {
        return rooms.count
    }
    subscript(i: Int) -> Room {
        return rooms[i]
    }
    func printNumberOfRooms() {
        print("房间号为 \(numberOfRooms)")
    }
    var address: Address?
}

// Room 定义一个name属性和一个设定room名的初始化器
class Room {
    let name: String
    init(name: String) { self.name = name }
}

// 模型中的最终类叫做Address
class Address {
    var buildingName: String?
    var buildingNumber: String?
    var street: String?
    func buildingIdentifier() -> String? {
        if (buildingName != nil) {
            return buildingName
        } else if (buildingNumber != nil) {
            return buildingNumber
        } else {
            return nil
        }
```

```
    }
}

let john = Person()
let johnsHouse = Residence()
johnsHouse.rooms.append(Room(name: "客厅"))
johnsHouse.rooms.append(Room(name: "厨房"))
john.residence = johnsHouse

if let firstRoomName = john.residence?[0].name {
    print("第一个房间名为\(firstRoomName)")
} else {
    print("无法检索到房间")
}
```

以上程序执行输出结果为：

```
第一个房间名为客厅
```

通过可选链接调用来访问下标

通过可选链接调用，我们可以用下标来对可选值进行读取或写入，并且判断下标调用是否成功。

实例

```
class Person {
    var residence: Residence?
}

// 定义了一个变量 rooms，它被初始化为一个Room[]类型的空数组
class Residence {
    var rooms = [Room]()
    var numberOfRooms: Int {
        return rooms.count
    }
    subscript(i: Int) -> Room {
        return rooms[i]
    }
    func printNumberOfRooms() {
        print("房间号为 \(numberOfRooms)")
    }
    var address: Address?
}

// Room 定义一个name属性和一个设定room名的初始化器
class Room {
```

```
    let name: String
    init(name: String) { self.name = name }
}

// 模型中的最终类叫做Address
class Address {
    var buildingName: String?
    var buildingNumber: String?
    var street: String?
    func buildingIdentifier() -> String? {
        if (buildingName != nil) {
            return buildingName
        } else if (buildingNumber != nil) {
            return buildingNumber
        } else {
            return nil
        }
    }
}

let john = Person()

let johnsHouse = Residence()
johnsHouse.rooms.append(Room(name: "客厅"))
johnsHouse.rooms.append(Room(name: "厨房"))
john.residence = johnsHouse

if let firstRoomName = john.residence?[0].name {
    print("第一个房间名为\(firstRoomName)")
} else {
    print("无法检索到房间")
}
```

以上程序执行输出结果为：

```
第一个房间名为客厅
```

访问可选类型的下标

如果下标返回可空类型值，比如Swift中Dictionary的key下标。可以在下标的闭合括号后面放一个问号来链接下标的可空返回值：

```
var testScores = ["Dave": [86, 82, 84], "Bev": [79, 94, 81]]
testScores["Dave"]?[0] = 91
testScores["Bev"]?[0]++
testScores["Brian"]?[0] = 72
// the "Dave" array is now [91, 82, 84] and the "Bev" array is now
```

上面的例子中定义了一个testScores数组，包含了两个键值对， 把String类型的key映射到一个整形数组。

这个例子用可选链接调用把"Dave"数组中第一个元素设为91，把"Bev"数组的第一个元素+1，然后尝试把"Brian"数组中的第一个元素设为72。

前两个调用是成功的，因为这两个key存在。但是key"Brian"在字典中不存在，所以第三个调用失败。

连接多层链接

你可以将多层可选链连接在一起，可以掘取模型内更下层的属性方法和下标脚本。然而多层可选链不能再添加比已经返回的可选值更多的层。

如果你试图通过可选链获得Int值，不论使用了多少层链接返回的总是Int?。 相似的，如果你试图通过可选链获得Int?值，不论使用了多少层链接返回的总是Int?。

实例1

下面的例子试图获取john的residence属性里的address的street属性。这里使用了两层可选链来联系residence和address属性，它们两者都是可选类型：

```
class Person {
    var residence: Residence?
}

// 定义了一个变量 rooms，它被初始化为一个Room[]类型的空数组
class Residence {
    var rooms = [Room]()
    var numberOfRooms: Int {
        return rooms.count
    }
    subscript(i: Int) -> Room {
        return rooms[i]
    }
    func printNumberOfRooms() {
        print("房间号为 \(numberOfRooms)")
    }
    var address: Address?
}

// Room 定义一个name属性和一个设定room名的初始化器
class Room {
    let name: String
    init(name: String) { self.name = name }
}

// 模型中的最终类叫做Address
class Address {
    var buildingName: String?
    var buildingNumber: String?
    var street: String?
    func buildingIdentifier() -> String? {
        if (buildingName != nil) {
            return buildingName
        } else if (buildingNumber != nil) {
            return buildingNumber
        } else {
            return nil
        }
    }
}

let john = Person()

if let johnsStreet = john.residence?.address?.street {
    print("John 的地址为 \(johnsStreet).")
} else {
    print("不能检索地址")
}
```

以上程序执行输出结果为：

```
不能检索地址
```

实例2

如果你为Address设定一个实例来作为john.residence.address的值，并为address的street属性设定一个实际值，你可以通过多层可选链来得到这个属性值。

```
class Person {
   var residence: Residence?
}

class Residence {

    var rooms = [Room]()
    var numberOfRooms: Int {
        return rooms.count
    }
    subscript(i: Int) -> Room {
        get{
            return rooms[i]
        }
        set {
            rooms[i] = newValue
        }
    }
    func printNumberOfRooms() {
        print("房间号为 \(numberOfRooms)")
    }
    var address: Address?
}

class Room {
    let name: String
    init(name: String) { self.name = name }
}

class Address {
    var buildingName: String?
    var buildingNumber: String?
    var street: String?
    func buildingIdentifier() -> String? {
        if (buildingName != nil) {
            return buildingName
        } else if (buildingNumber != nil) {
            return buildingNumber
        } else {
            return nil
        }
    }
}
```

```
let john = Person()
john.residence?[0] = Room(name: "浴室")

let johnsHouse = Residence()
johnsHouse.rooms.append(Room(name: "客厅"))
johnsHouse.rooms.append(Room(name: "厨房"))
john.residence = johnsHouse

if let firstRoomName = john.residence?[0].name {
    print("第一个房间是\(firstRoomName)")
} else {
    print("无法检索房间")
}
```

以上实例输出结果为：

```
第一个房间是客厅
```

对返回可选值的函数进行链接

我们还可以通过可选链接来调用返回可空值的方法，并且可以继续对可选值进行链接。

实例

```
class Person {
    var residence: Residence?
}

// 定义了一个变量 rooms，它被初始化为一个Room[]类型的空数组
class Residence {
    var rooms = [Room]()
    var numberOfRooms: Int {
        return rooms.count
    }
    subscript(i: Int) -> Room {
        return rooms[i]
    }
    func printNumberOfRooms() {
        print("房间号为 \(numberOfRooms)")
    }
    var address: Address?
}

// Room 定义一个name属性和一个设定room名的初始化器
class Room {
    let name: String
    init(name: String) { self.name = name }
}

// 模型中的最终类叫做Address
class Address {
    var buildingName: String?
    var buildingNumber: String?
    var street: String?
    func buildingIdentifier() -> String? {
        if (buildingName != nil) {
            return buildingName
        } else if (buildingNumber != nil) {
            return buildingNumber
        } else {
            return nil
        }
    }
}

let john = Person()

if john.residence?.printNumberOfRooms() != nil {
    print("指定了房间号)")
}  else {
    print("未指定房间号")
}
```

以上程序执行输出结果为：

```
未指定房间号
```

Swift 自动引用计数（ARC）

Swift 使用自动引用计数（ARC）这一机制来跟踪和管理应用程序的内存

通常情况下我们不需要去手动释放内存，因为 ARC 会在类的实例不再被使用时，自动释放其占用的内存。

但在有些时候我们还是需要在代码中实现内存管理。

ARC 功能

- 当每次使用 init() 方法创建一个类的新的实例的时候，ARC 会分配一大块内存用来储存实例的信息。
- 内存中会包含实例的类型信息，以及这个实例所有相关属性的值。
- 当实例不再被使用时，ARC 释放实例所占用的内存，并让释放的内存能挪作他用。
- 为了确保使用中的实例不会被销毁，ARC 会跟踪和计算每一个实例正在被多少属性，常量和变量所引用。
- 实例赋值给属性、常量或变量，它们都会创建此实例的强引用，只要强引用还在，实例是不允许被销毁的。

ARC 实例

```
class Person {
    let name: String
    init(name: String) {
        self.name = name
        print("\(name) 开始初始化")
    }
    deinit {
        print("\(name) 被析构")
    }
}

// 值会被自动初始化为nil，目前还不会引用到Person类的实例
var reference1: Person?
var reference2: Person?
var reference3: Person?

// 创建Person类的新实例
reference1 = Person(name: "Runoob")

//赋值给其他两个变量，该实例又会多出两个强引用
reference2 = reference1
reference3 = reference1

//断开第一个强引用
reference1 = nil
//断开第二个强引用
reference2 = nil
//断开第三个强引用，并调用析构函数
reference3 = nil
```

以上程序执行输出结果为：

```
Runoob 开始初始化
Runoob 被析构
```

类实例之间的循环强引用

在上面的例子中，ARC 会跟踪你所新创建的 Person 实例的引用数量，并且会在 Person 实例不再被需要时销毁它。

然而，我们可能会写出这样的代码，一个类永远不会有0个强引用。这种情况发生在两个类实例互相保持对方的强引用，并让对方不被销毁。这就是所谓的循环强引用。

实例

下面展示了一个不经意产生循环强引用的例子。例子定义了两个类：Person和Apartment，用来建模公寓和它其中的居民:

```
class Person {
    let name: String
    init(name: String) { self.name = name }
    var apartment: Apartment?
    deinit { print("\(name) 被析构") }
}

class Apartment {
    let number: Int
    init(number: Int) { self.number = number }
    var tenant: Person?
    deinit { print("Apartment #\(number) 被析构") }
}

// 两个变量都被初始化为nil
var runoob: Person?
var number73: Apartment?

// 赋值
runoob = Person(name: "Runoob")
number73 = Apartment(number: 73)

// 意感叹号是用来展开和访问可选变量 runoob 和 number73 中的实例
// 循环强引用被创建
runoob!.apartment = number73
number73!.tenant = runoob

// 断开 runoob 和 number73 变量所持有的强引用时，引用计数并不会降为 0，实例
// 注意，当你把这两个变量设为nil时，没有任何一个析构函数被调用。
// 强引用循环阻止了Person和Apartment类实例的销毁，并在你的应用程序中造成了内
runoob = nil
number73 = nil
```

解决实例之间的循环强引用

Swift 提供了两种办法用来解决你在使用类的属性时所遇到的循环强引用问题：

- 弱引用
- 无主引用

弱引用和无主引用允许循环引用中的一个实例引用另外一个实例而不保持强引用。这样实例能够互相引用而不产生循环强引用。

对于生命周期中会变为nil的实例使用弱引用。相反的，对于初始化赋值后再也不会被赋值为nil的实例，使用无主引用。

弱引用实例

```
class Module {
    let name: String
    init(name: String) { self.name = name }
    var sub: SubModule?
    deinit { print("\(name) 主模块") }
}

class SubModule {
    let number: Int

    init(number: Int) { self.number = number }

    weak var topic: Module?

    deinit { print("子模块 topic 数为 \(number)") }
}

var toc: Module?
var list: SubModule?
toc = Module(name: "ARC")
list = SubModule(number: 4)
toc!.sub = list
list!.topic = toc

toc = nil
list = nil
```

以上程序执行输出结果为：

```
ARC 主模块
子模块 topic 数为 4
```

无主引用实例

```
class Student {
    let name: String
    var section: Marks?

    init(name: String) {
        self.name = name
    }

    deinit { print("\(name)") }
}
class Marks {
    let marks: Int
    unowned let stname: Student

    init(marks: Int, stname: Student) {
        self.marks = marks
        self.stname = stname
    }

    deinit { print("学生的分数为 \(marks)") }
}

var module: Student?
module = Student(name: "ARC")
module!.section = Marks(marks: 98, stname: module!)
module = nil
```

以上程序执行输出结果为：

```
ARC
学生的分数为 98
```

闭包引起的循环强引用

循环强引用还会发生在当你将一个闭包赋值给类实例的某个属性，并且这个闭包体中又使用了实例。这个闭包体中可能访问了实例的某个属性，例如self.someProperty，或者闭包中调用了实例的某个方法，例如self.someMethod。这两种情况都导致了闭包 "捕获" self，从而产生了循环强引用。

实例

下面的例子为你展示了当一个闭包引用了self后是如何产生一个循环强引用的。例子中定义了一个叫HTMLElement的类，用一种简单的模型表示 HTML 中的一个单独的元素：

```
class HTMLElement {

    let name: String
    let text: String?

    lazy var asHTML: () -> String = {
        if let text = self.text {
            return "<\(self.name)>\(text)</\(self.name)>"
        } else {
            return "<\(self.name) />"
        }
    }

    init(name: String, text: String? = nil) {
        self.name = name
        self.text = text
    }

    deinit {
        print("\(name) is being deinitialized")
    }

}

// 创建实例并打印信息
var paragraph: HTMLElement? = HTMLElement(name: "p", text: "hello,
print(paragraph!.asHTML())
```

HTMLElement 类产生了类实例和 asHTML 默认值的闭包之间的循环强引用。

实例的 asHTML 属性持有闭包的强引用。但是，闭包在其闭包体内使用了self（引用了self.name和self.text），因此闭包捕获了self，这意味着闭包又反过来持有了HTMLElement实例的强引用。这样两个对象就产生了循环强引用。

解决闭包引起的循环强引用:在定义闭包时同时定义捕获列表作为闭包的一部分，通过这种方式可以解决闭包和类实例之间的循环强引用。

弱引用和无主引用

当闭包和捕获的实例总是互相引用时并且总是同时销毁时，将闭包内的捕获定义为无主引用。

相反的，当捕获引用有时可能会是nil时，将闭包内的捕获定义为弱引用。

如果捕获的引用绝对不会置为nil，应该用无主引用，而不是弱引用。

实例

前面的HTMLElement例子中，无主引用是正确的解决循环强引用的方法。这样编写HTMLElement类来避免循环强引用：

```
class HTMLElement {

    let name: String
    let text: String?

    lazy var asHTML: () -> String = {
        [unowned self] in
        if let text = self.text {
            return "<\(self.name)>\(text)</\(self.name)>"
        } else {
            return "<\(self.name) />"
        }
    }

    init(name: String, text: String? = nil) {
        self.name = name
        self.text = text
    }

    deinit {
        print("\(name) 被析构")
    }

}

//创建并打印HTMLElement实例
var paragraph: HTMLElement? = HTMLElement(name: "p", text: "hello,
print(paragraph!.asHTML())

// HTMLElement实例将会被销毁，并能看到它的析构函数打印出的消息
paragraph = nil
```

以上程序执行输出结果为：

```
<p>hello, world</p>
p 被析构
```

Swift 类型转换

Swift 语言类型转换可以判断实例的类型。也可以用于检测实例类型是否属于其父类或者子类的实例。

Swift 中类型转换使用 is 和 as 操作符实现，is 用于检测值的类型，as 用于转换类型。

类型转换也可以用来检查一个类是否实现了某个协议。

定义一个类层次

类型转换用于检测实例类型是否属于特定的实例类型。

你可以将它用在类和子类的层次结构上，检查特定类实例的类型并且转换这个类实例的类型成为这个层次结构中的其他类型。

实例如下：

```
class Subjects {
    var physics: String
    init(physics: String) {
        self.physics = physics
    }
}

class Chemistry: Subjects {
    var equations: String
    init(physics: String, equations: String) {
        self.equations = equations
        super.init(physics: physics)
    }
}

class Maths: Subjects {
    var formulae: String
    init(physics: String, formulae: String) {
        self.formulae = formulae
        super.init(physics: physics)
    }
}

let sa = [
    Chemistry(physics: "固体物理", equations: "赫兹"),
    Maths(physics: "流体动力学", formulae: "千兆赫")]

let samplechem = Chemistry(physics: "固体物理", equations: "赫兹")
print("实例物理学是: \(samplechem.physics)")
print("实例方程式: \(samplechem.equations)")

let samplemaths = Maths(physics: "流体动力学", formulae: "千兆赫")
print("实例物理学是: \(samplemaths.physics)")
print("实例公式是: \(samplemaths.formulae)")
```

以上程序执行输出结果为：

```
实例物理学是: 固体物理
实例方程式: 赫兹
实例物理学是: 流体动力学
实例公式是: 千兆赫
```

检查类型

类型检查使用 **is** 关键字。

操作符 **is** 来检查一个实例是否属于特定子类型。若实例属于那个子类型，类型检查操作符返回 true，否则返回 false。

```
class Subjects {
    var physics: String
    init(physics: String) {
        self.physics = physics
    }
}

class Chemistry: Subjects {
    var equations: String
    init(physics: String, equations: String) {
        self.equations = equations
        super.init(physics: physics)
    }
}

class Maths: Subjects {
    var formulae: String
    init(physics: String, formulae: String) {
        self.formulae = formulae
        super.init(physics: physics)
    }
}

let sa = [
    Chemistry(physics: "固体物理", equations: "赫兹"),
    Maths(physics: "流体动力学", formulae: "千兆赫"),
    Chemistry(physics: "热物理学", equations: "分贝"),
    Maths(physics: "天体物理学", formulae: "兆赫"),
    Maths(physics: "微分方程", formulae: "余弦级数")]

let samplechem = Chemistry(physics: "固体物理", equations: "赫兹")
print("实例物理学是: \(samplechem.physics)")
print("实例方程式: \(samplechem.equations)")

let samplemaths = Maths(physics: "流体动力学", formulae: "千兆赫")
print("实例物理学是: \(samplemaths.physics)")
print("实例公式是: \(samplemaths.formulae)")

var chemCount = 0
var mathsCount = 0
for item in sa {
    // 如果是一个 Chemistry 类型的实例，返回 true，相反返回 false。
    if item is Chemistry {
        ++chemCount
    } else if item is Maths {
        ++mathsCount
    }
}

print("化学科目包含 \(chemCount) 个主题，数学包含 \(mathsCount) 个主题")
```

以上程序执行输出结果为：

```
实例物理学是: 固体物理
实例方程式: 赫兹
实例物理学是: 流体动力学
实例公式是: 千兆赫
化学科目包含 2 个主题，数学包含 3 个主题
```

向下转型

向下转型，用类型转换操作符(as? 或 as!)

当你不确定向下转型可以成功时，用类型转换的条件形式(as?)。条件形式的类型转换总是返回一个可选值（optional value），并且若下转是不可能的，可选值将是nil。

只有你可以确定向下转型一定会成功时，才使用强制形式(as!)。当你试图向下转型为一个不正确的类型时，强制形式的类型转换会触发一个运行时错误。

```
class Subjects {
    var physics: String
    init(physics: String) {
        self.physics = physics
    }
}

class Chemistry: Subjects {
    var equations: String
    init(physics: String, equations: String) {
        self.equations = equations
        super.init(physics: physics)
    }
}

class Maths: Subjects {
    var formulae: String
    init(physics: String, formulae: String) {
        self.formulae = formulae
        super.init(physics: physics)
    }
}

let sa = [
    Chemistry(physics: "固体物理", equations: "赫兹"),
    Maths(physics: "流体动力学", formulae: "千兆赫"),
    Chemistry(physics: "热物理学", equations: "分贝"),
    Maths(physics: "天体物理学", formulae: "兆赫"),
    Maths(physics: "微分方程", formulae: "余弦级数")]

let samplechem = Chemistry(physics: "固体物理", equations: "赫兹")
print("实例物理学是: \(samplechem.physics)")
print("实例方程式: \(samplechem.equations)")

let samplemaths = Maths(physics: "流体动力学", formulae: "千兆赫")
print("实例物理学是: \(samplemaths.physics)")
print("实例公式是: \(samplemaths.formulae)")

var chemCount = 0
var mathsCount = 0

for item in sa {
    // 类型转换的条件形式
    if let show = item as? Chemistry {
        print("化学主题是: '\(show.physics)', \(show.equations)")
        // 强制形式
    } else if let example = item as? Maths {
        print("数学主题是: '\(example.physics)',  \(example.formulae
    }
}
```

以上程序执行输出结果为：

```
实例物理学是: 固体物理
实例方程式: 赫兹
实例物理学是: 流体动力学
实例公式是: 千兆赫
化学主题是: '固体物理', 赫兹
数学主题是: '流体动力学',  千兆赫
化学主题是: '热物理学', 分贝
数学主题是: '天体物理学',  兆赫
数学主题是: '微分方程',  余弦级数
```

Any和AnyObject的类型转换

Swift为不确定类型提供了两种特殊类型别名：

- `AnyObject` 可以代表任何class类型的实例。
- `Any` 可以表示任何类型，包括方法类型（function types）。

> 注意：
>
> 只有当你明确的需要它的行为和功能时才使用 `Any` 和 `AnyObject` 。在你的代码里使用你期望的明确的类型总是更好的。

Any 实例

```
class Subjects {
    var physics: String
    init(physics: String) {
        self.physics = physics
    }
}

class Chemistry: Subjects {
    var equations: String
    init(physics: String, equations: String) {
        self.equations = equations
        super.init(physics: physics)
    }
}

class Maths: Subjects {
    var formulae: String
    init(physics: String, formulae: String) {
        self.formulae = formulae
        super.init(physics: physics)
    }
}
```

```
let sa = [
    Chemistry(physics: "固体物理", equations: "赫兹"),
    Maths(physics: "流体动力学", formulae: "千兆赫"),
    Chemistry(physics: "热物理学", equations: "分贝"),
    Maths(physics: "天体物理学", formulae: "兆赫"),
    Maths(physics: "微分方程", formulae: "余弦级数")]

let samplechem = Chemistry(physics: "固体物理", equations: "赫兹")
print("实例物理学是: \(samplechem.physics)")
print("实例方程式: \(samplechem.equations)")

let samplemaths = Maths(physics: "流体动力学", formulae: "千兆赫")
print("实例物理学是: \(samplemaths.physics)")
print("实例公式是: \(samplemaths.formulae)")

var chemCount = 0
var mathsCount = 0

for item in sa {
    // 类型转换的条件形式
    if let show = item as? Chemistry {
        print("化学主题是: '\(show.physics)', \(show.equations)")
        // 强制形式
    } else if let example = item as? Maths {
        print("数学主题是: '\(example.physics)',  \(example.formulae
    }
}

// 可以存储Any类型的数组 exampleany
var exampleany = [Any]()

exampleany.append(12)
exampleany.append(3.14159)
exampleany.append("Any 实例")
exampleany.append(Chemistry(physics: "固体物理", equations: "兆赫"))

for item2 in exampleany {
    switch item2 {
    case let someInt as Int:
        print("整型值为 \(someInt)")
    case let someDouble as Double where someDouble > 0:
        print("Pi 值为 \(someDouble)")
    case let someString as String:
        print("\(someString)")
    case let phy as Chemistry:
        print("主题 '\(phy.physics)', \(phy.equations)")
    default:
        print("None")
    }
}
```

以上程序执行输出结果为：

```
实例物理学是: 固体物理
实例方程式: 赫兹
实例物理学是: 流体动力学
实例公式是: 千兆赫
化学主题是: '固体物理', 赫兹
数学主题是: '流体动力学',  千兆赫
化学主题是: '热物理学', 分贝
数学主题是: '天体物理学',  兆赫
数学主题是: '微分方程',  余弦级数
整型值为 12
Pi 值为 3.14159
Any 实例
主题 '固体物理', 兆赫
```

AnyObject 实例

```
class Subjects {
    var physics: String
    init(physics: String) {
        self.physics = physics
    }
}

class Chemistry: Subjects {
    var equations: String
    init(physics: String, equations: String) {
        self.equations = equations
        super.init(physics: physics)
    }
}

class Maths: Subjects {
    var formulae: String
    init(physics: String, formulae: String) {
        self.formulae = formulae
        super.init(physics: physics)
    }
}

// [AnyObject] 类型的数组
let saprint: [AnyObject] = [
    Chemistry(physics: "固体物理", equations: "赫兹"),
    Maths(physics: "流体动力学", formulae: "千兆赫"),
    Chemistry(physics: "热物理学", equations: "分贝"),
    Maths(physics: "天体物理学", formulae: "兆赫"),
    Maths(physics: "微分方程", formulae: "余弦级数")]

let samplechem = Chemistry(physics: "固体物理", equations: "赫兹")
```

```
print("实例物理学是: \(samplechem.physics)")
print("实例方程式: \(samplechem.equations)")

let samplemaths = Maths(physics: "流体动力学", formulae: "千兆赫")
print("实例物理学是: \(samplemaths.physics)")
print("实例公式是: \(samplemaths.formulae)")

var chemCount = 0
var mathsCount = 0

for item in saprint {
    // 类型转换的条件形式
    if let show = item as? Chemistry {
        print("化学主题是: '\(show.physics)', \(show.equations)")
        // 强制形式
    } else if let example = item as? Maths {
        print("数学主题是: '\(example.physics)',  \(example.formulae
    }
}

var exampleany = [Any]()
exampleany.append(12)
exampleany.append(3.14159)
exampleany.append("Any 实例")
exampleany.append(Chemistry(physics: "固体物理", equations: "兆赫"))

for item2 in exampleany {
    switch item2 {
    case let someInt as Int:
        print("整型值为 \(someInt)")
    case let someDouble as Double where someDouble > 0:
        print("Pi 值为 \(someDouble)")
    case let someString as String:
        print("\(someString)")
    case let phy as Chemistry:
        print("主题 '\(phy.physics)', \(phy.equations)")
    default:
        print("None")
    }
}
```

以上程序执行输出结果为：

```
实例物理学是: 固体物理
实例方程式: 赫兹
实例物理学是: 流体动力学
实例公式是: 千兆赫
化学主题是: '固体物理', 赫兹
数学主题是: '流体动力学',  千兆赫
化学主题是: '热物理学', 分贝
数学主题是: '天体物理学',  兆赫
数学主题是: '微分方程',  余弦级数
整型值为 12
Pi 值为 3.14159
Any 实例
主题 '固体物理', 兆赫
```

在一个switch语句的case中使用强制形式的类型转换操作符（as, 而不是 as?）来检查和转换到一个明确的类型。

Swift 扩展

扩展就是向一个已有的类、结构体或枚举类型添加新功能。

扩展可以对一个类型添加新的功能，但是不能重写已有的功能。

Swift 中的扩展可以：

- 添加计算型属性和计算型静态属性
- 定义实例方法和类型方法
- 提供新的构造器
- 定义下标
- 定义和使用新的嵌套类型
- 使一个已有类型符合某个协议

语法

扩展声明使用关键字 **extension**：

```
extension SomeType  {  // 加到SomeType的新功能写到这里  }
```

一个扩展可以扩展一个已有类型，使其能够适配一个或多个协议，语法格式如下：

```
extension SomeType:  SomeProtocol,  AnotherProctocol  {  // 协议实现
```

计算型属性

扩展可以向已有类型添加计算型实例属性和计算型类型属性。

实例

下面的例子向 Int 类型添加了 5 个计算型实例属性并扩展其功能：

```
extension Int  {  var add:  Int  {return  self  +  100  }  var  sub
```

以上程序执行输出结果为：

```
加法运算后的值：103  减法运算后的值：110  乘法运算后的值：390  除法运算后的值
```

构造器

扩展可以向已有类型添加新的构造器。

这可以让你扩展其它类型，将你自己的定制类型作为构造器参数，或者提供该类型的原始实现中没有包含的额外初始化选项。

扩展可以向类中添加新的便利构造器 init()，但是它们不能向类中添加新的指定构造器或析构函数 deinit() 。

```
struct sum {  var num1 =  100, num2 =  200  }  struct diff {  var n
        _ = y.no1 + y.no2 }  }  let a = sum(num1:  100, num2:  200)
```

以上程序执行输出结果为：

```
mult 模块内  (100,  200) mult 模块内  (200,  100) mult 模块内  (300,
```

方法

扩展可以向已有类型添加新的实例方法和类型方法。

下面的例子向Int类型添加一个名为 topics 的新实例方法：

```
extension sum Int  { func topics(summation:  ()  ->  ())  {  for _ in
```

以上程序执行输出结果为：

```
扩展模块内  扩展模块内  扩展模块内  扩展模块内  内型转换模块内  内型转换模块内
```

这个 `topics` 方法使用了一个 `() -> ()` 类型的单参数，表明函数没有参数而且没有返回值。

定义该扩展之后，你就可以对任意整数调用 `topics` 方法,实现的功能则是多次执行某任务：

可变实例方法

通过扩展添加的实例方法也可以修改该实例本身。

结构体和枚举类型中修改self或其属性的方法必须将该实例方法标注为mutating，正如来自原始实现的修改方法一样。

实例

下面的例子向 Swift 的 Double 类型添加了一个新的名为 square 的修改方法，来实现一个原始值的平方计算：

```
extension Double  { mutating func square()  {  let pi =  3.1415  se
```

以上程序执行输出结果为：

```
圆的面积为:  34.210935  圆的面积为:  105.68006  圆的面积为:  45464.0707
```

下标

扩展可以向一个已有类型添加新下标。

实例

以下例子向 Swift 内建类型Int添加了一个整型下标。该下标[n]返回十进制数字

```
extension Int  { subscript(var multtable:  Int)  ->  Int  {  var nc
```

以上程序执行输出结果为：

```
2  6  5
```

嵌套类型

扩展可以向已有的类、结构体和枚举添加新的嵌套类型：

```
extension Int  {  enum calc {  case add case  sub  case mult case
```

以上程序执行输出结果为：

```
10  20  30  40  50  50
```

Swift 协议

协议规定了用来实现某一特定功能所必需的方法和属性。

任意能够满足协议要求的类型被称为遵循(conform)这个协议。

类，结构体或枚举类型都可以遵循协议，并提供具体实现来完成协议定义的方法和功能。

语法

协议的语法格式如下：

```
protocol SomeProtocol {
    // 协议内容
}
```

要使类遵循某个协议，需要在类型名称后加上协议名称，中间以冒号:分隔，作为类型定义的一部分。遵循多个协议时，各协议之间用逗号,分隔。

```
struct SomeStructure: FirstProtocol, AnotherProtocol {
    // 结构体内容
}
```

如果类在遵循协议的同时拥有父类，应该将父类名放在协议名之前，以逗号分隔。

```
class SomeClass: SomeSuperClass, FirstProtocol, AnotherProtocol {
    // 类的内容
}
```

对属性的规定

协议用于指定特定的实例属性或类属性，而不用指定是存储型属性或计算型属性。此外还必须指明是只读的还是可读可写的。

协议中的通常用var来声明变量属性，在类型声明后加上{ set get }来表示属性是可读可写的，只读属性则用{ get }来表示。

```
protocol classa {

    var marks: Int { get set }
    var result: Bool { get }

    func attendance() -> String
    func markssecured() -> String

}

protocol classb: classa {

    var present: Bool { get set }
    var subject: String { get set }
    var stname: String { get set }

}

class classc: classb {
    var marks = 96
    let result = true
    var present = false
    var subject = "Swift 协议"
    var stname = "Protocols"

    func attendance() -> String {
        return "The \(stname) has secured 99% attendance"
    }

    func markssecured() -> String {
        return "\(stname) has scored \(marks)"
    }
}

let studdet = classc()
studdet.stname = "Swift"
studdet.marks = 98
studdet.markssecured()

print(studdet.marks)
print(studdet.result)
print(studdet.present)
print(studdet.subject)
print(studdet.stname)
```

以上程序执行输出结果为：

```
98
true
false
Swift 协议
Swift
```

对 Mutating 方法的规定

有时需要在方法中改变它的实例。

例如，值类型(结构体，枚举)的实例方法中，将mutating关键字作为函数的前缀，写在func之前，表示可以在该方法中修改它所属的实例及其实例属性的值。

```
protocol daysofaweek {
    mutating func show()
}

enum days: daysofaweek {
    case sun, mon, tue, wed, thurs, fri, sat
    mutating func show() {
        switch self {
        case sun:
            self = sun
            print("Sunday")
        case mon:
            self = mon
            print("Monday")
        case tue:
            self = tue
            print("Tuesday")
        case wed:
            self = wed
            print("Wednesday")
        case mon:
            self = thurs
            print("Thursday")
        case tue:
            self = fri
            print("Friday")
        case sat:
            self = sat
            print("Saturday")
        default:
            print("NO Such Day")
        }
    }
}

var res = days.wed
res.show()
```

以上程序执行输出结果为：

```
Wednesday
```

对构造器的规定

协议可以要求它的遵循者实现指定的构造器。

你可以像书写普通的构造器那样，在协议的定义里写下构造器的声明，但不需要写花括号和构造器的实体，语法如下：

```
protocol SomeProtocol {
   init(someParameter: Int)
}
```

实例

```
protocol tcpprotocol {
   init(aprot: Int)
}
```

协议构造器规定在类中的实现

你可以在遵循该协议的类中实现构造器，并指定其为类的指定构造器或者便利构造器。在这两种情况下，你都必须给构造器实现标上"required"修饰符：

```
class SomeClass: SomeProtocol {
   required init(someParameter: Int) {
      // 构造器实现
   }
}

protocol tcpprotocol {
   init(aprot: Int)
}

class tcpClass: tcpprotocol {
   required init(aprot: Int) {
   }
}
```

使用required修饰符可以保证：所有的遵循该协议的子类，同样能为构造器规定提供一个显式的实现或继承实现。

如果一个子类重写了父类的指定构造器，并且该构造器遵循了某个协议的规定，那么该构造器的实现需要被同时标示required和override修饰符：

```
protocol tcpprotocol {
    init(no1: Int)
}

class mainClass {
    var no1: Int // 局部变量
    init(no1: Int) {
        self.no1 = no1 // 初始化
    }
}

class subClass: mainClass, tcpprotocol {
    var no2: Int
    init(no1: Int, no2 : Int) {
        self.no2 = no2
        super.init(no1:no1)
    }
    // 因为遵循协议，需要加上"required"; 因为继承自父类，需要加上"override
    required override convenience init(no1: Int)  {
        self.init(no1:no1, no2:0)
    }
}
let res = mainClass(no1: 20)
let show = subClass(no1: 30, no2: 50)

print("res is: \(res.no1)")
print("res is: \(show.no1)")
print("res is: \(show.no2)")
```

以上程序执行输出结果为：

```
res is: 20
res is: 30
res is: 50
```

协议类型

尽管协议本身并不实现任何功能，但是协议可以被当做类型来使用。

协议可以像其他普通类型一样使用，使用场景:

- 作为函数、方法或构造器中的参数类型或返回值类型
- 作为常量、变量或属性的类型
- 作为数组、字典或其他容器中的元素类型

实例

```
protocol Generator {
    typealias members
    func next() -> members?
}

var items = [10,20,30].generate()
while let x = items.next() {
    print(x)
}

for lists in [1,2,3].map( {i in i*5}) {
    print(lists)
}

print([100,200,300])
print([1,2,3].map({i in i*10}))
```

以上程序执行输出结果为：

```
10
20
30
5
10
15
[100, 200, 300]
[10, 20, 30]
```

在扩展中添加协议成员

我们可以可以通过扩展来扩充已存在类型(类，结构体，枚举等)。

扩展可以为已存在的类型添加属性，方法，下标脚本，协议等成员。

```
protocol AgeClasificationProtocol {
   var age: Int { get }
   func agetype() -> String
}

class Person {
   let firstname: String
   let lastname: String
   var age: Int
   init(firstname: String, lastname: String) {
      self.firstname = firstname
      self.lastname = lastname
      self.age = 10
   }
}

extension Person : AgeClasificationProtocol {
   func fullname() -> String {
      var c: String
      c = firstname + " " + lastname
      return c
   }

   func agetype() -> String {
      switch age {
      case 0...2:
         return "Baby"
      case 2...12:
         return "Child"
      case 13...19:
         return "Teenager"
      case let x where x > 65:
         return "Elderly"
      default:
         return "Normal"
      }
   }
}
```

协议的继承

协议能够继承一个或多个其他协议，可以在继承的协议基础上增加新的内容要求。

```
protocol InheritingProtocol: SomeProtocol, AnotherProtocol {
    // 协议定义
}
```

实例

```
protocol Classa {
    var no1: Int { get set }
    func calc(sum: Int)
}

protocol Result {
    func print(target: Classa)
}

class Student2: Result {
    func print(target: Classa) {
        target.calc(1)
    }
}

class Classb: Result {
    func print(target: Classa) {
        target.calc(5)
    }
}

class Student: Classa {
    var no1: Int = 10

    func calc(sum: Int) {
        no1 -= sum
        print("学生尝试 \(sum) 次通过")

        if no1 <= 0 {
            print("学生缺席考试")
        }
    }
}

class Player {
    var stmark: Result!

    init(stmark: Result) {
        self.stmark = stmark
    }

    func print(target: Classa) {
        stmark.print(target)
    }
}

var marks = Player(stmark: Student2())
var marksec = Student()

marks.print(marksec)
```

```
marks.print(marksec)
marks.print(marksec)
marks.stmark = Classb()
marks.print(marksec)
marks.print(marksec)
marks.print(marksec)
```

以上程序执行输出结果为：

```
学生尝试 1 次通过
学生尝试 1 次通过
学生尝试 1 次通过
学生尝试 5 次通过
学生尝试 5 次通过
学生缺席考试
学生尝试 5 次通过
学生缺席考试
```

类专属协议

你可以在协议的继承列表中,通过添加class关键字,限制协议只能适配到类（class）类型。

该class关键字必须是第一个出现在协议的继承列表中，其后，才是其他继承协议。格式如下：

```
protocol SomeClassOnlyProtocol: class, SomeInheritedProtocol {
    // 协议定义
}
```

实例

```
protocol TcpProtocol {
    init(no1: Int)
}

class MainClass {
    var no1: Int // 局部变量
    init(no1: Int) {
        self.no1 = no1 // 初始化
    }
}

class SubClass: MainClass, TcpProtocol {
    var no2: Int
    init(no1: Int, no2 : Int) {
        self.no2 = no2
        super.init(no1:no1)
    }
    // 因为遵循协议，需要加上"required"; 因为继承自父类，需要加上"override
    required override convenience init(no1: Int)  {
        self.init(no1:no1, no2:0)
    }
}

let res = MainClass(no1: 20)
let show = SubClass(no1: 30, no2: 50)

print("res is: \(res.no1)")
print("res is: \(show.no1)")
print("res is: \(show.no2)")
```

以上程序执行输出结果为：

```
res is: 20
res is: 30
res is: 50
```

协议合成

Swift 支持合成多个协议，这在我们需要同时遵循多个协议时非常有用。

语法格式如下：

```
protocol<SomeProtocol, AnotherProtocol>
```

实例

```
protocol Stname {
    var name: String { get }
}

protocol Stage {
    var age: Int { get }
}

struct Person: Stname, Stage {
    var name: String
    var age: Int
}

func show(celebrator: protocol<Stname, Stage>) {
    print("\(celebrator.name) is \(celebrator.age) years old")
}

let studname = Person(name: "Priya", age: 21)
print(studname)

let stud = Person(name: "Rehan", age: 29)
print(stud)

let student = Person(name: "Roshan", age: 19)
print(student)
```

以上程序执行输出结果为：

```
Person(name: "Priya", age: 21)
Person(name: "Rehan", age: 29)
Person(name: "Roshan", age: 19)
```

检验协议的一致性

你可以使用is和as操作符来检查是否遵循某一协议或强制转化为某一类型。

- `is` 操作符用来检查实例是否 `遵循` 了某个 `协议` 。
- `as?` 返回一个可选值，当实例 `遵循` 协议时，返回该协议类型;否则返回 `nil` 。
- `as` 用以强制向下转型，如果强转失败，会引起运行时错误。

实例

下面的例子定义了一个 HasArea 的协议，要求有一个Double类型可读的 area：

```
protocol HasArea {
    var area: Double { get }
}

// 定义了Circle类，都遵循了HasArea协议
class Circle: HasArea {
    let pi = 3.1415927
    var radius: Double
    var area: Double { return pi * radius * radius }
    init(radius: Double) { self.radius = radius }
}

// 定义了Country类，都遵循了HasArea协议
class Country: HasArea {
    var area: Double
    init(area: Double) { self.area = area }
}

// Animal是一个没有实现HasArea协议的类
class Animal {
    var legs: Int
    init(legs: Int) { self.legs = legs }
}

let objects: [AnyObject] = [
    Circle(radius: 2.0),
    Country(area: 243_610),
    Animal(legs: 4)
]

for object in objects {
    // 对迭代出的每一个元素进行检查，看它是否遵循了HasArea协议
    if let objectWithArea = object as? HasArea {
        print("面积为 \(objectWithArea.area)")
    } else {
        print("没有面积")
    }
}
```

以上程序执行输出结果为：

```
面积为 12.5663708
面积为 243610.0
没有面积
```

Swift 泛型

Swift 提供了泛型让你写出灵活且可重用的函数和类型。

Swift 标准库是通过泛型代码构建出来的。

Swift 的数组和字典类型都是泛型集。

你可以创建一个Int数组，也可创建一个String数组，或者甚至于可以是任何其他 Swift 的类型数据数组。

以下实例是一个非泛型函数 exchange 用来交换两个 Int 值：

```
// 定义一个交换两个变量的函数
func exchange(inout a: Int, inout b: Int) {
    let temp = a
    a = b
    b = temp
}

var numb1 = 100
var numb2 = 200

print("交换前数据: \(numb1) 和 \(numb2)")
exchange(&numb1, b: &numb2)
print("交换后数据: \(numb1) 和 \(numb2)")
```

以上程序执行输出结果为：

```
交换前数据: 100 和 200
交换后数据: 200 和 100
```

泛型函数可以访问任何类型，如 Int 或 String。

以下实例是一个泛型函数 exchange 用来交换两个 Int 和 String 值：

```
func exchange<T>(inout a: T, inout b: T) {
    let temp = a
    a = b
    b = temp
}

var numb1 = 100
var numb2 = 200

print("交换前数据:  \(numb1) 和 \(numb2)")
exchange(&numb1, b: &numb2)
print("交换后数据: \(numb1) 和 \(numb2)")

var str1 = "A"
var str2 = "B"

print("交换前数据:  \(str1) 和 \(str2)")
exchange(&str1, b: &str2)
print("交换后数据: \(str1) 和 \(str2)")
```

以上程序执行输出结果为：

```
交换前数据:  100 和 200
交换后数据: 200 和 100
交换前数据:  A 和 B
交换后数据: B 和 A
```

这个函数的泛型版本使用了占位类型名字（通常此情况下用字母T来表示）来代替实际类型名（如Int、String或Double）。占位类型名没有提示T必须是什么类型，但是它提示了a和b必须是同一类型T，而不管T表示什么类型。只有 exchange(::)函数在每次调用时所传入的实际类型才能决定T所代表的类型。

另外一个不同之处在于这个泛型函数名后面跟着的占位类型名字（T）是用尖括号括起来的（<t>）。这个尖括号告诉 Swift 那个T是 exchange(::)函数所定义的一个类型。因为T是一个占位命名类型，Swift 不会去查找命名为T的实际类型。</t>

泛型类型

Swift 允许你定义你自己的泛型类型。

自定义类、结构体和枚举作用于任何类型，如同Array和Dictionary的用法。

```
struct TOS<T> {
    var items = [T]()
    mutating func push(item: T) {
        items.append(item)
    }

    mutating func pop() -> T {
        return items.removeLast()
    }
}

var tos = TOS<String>()
tos.push("Swift")
print(tos.items)

tos.push("泛型")
print(tos.items)

tos.push("类型参数")
print(tos.items)

tos.push("类型参数名")
print(tos.items)

let deletetos = tos.pop()
```

以上程序执行输出结果为：

```
["Swift"]
["Swift", "泛型"]
["Swift", "泛型", "类型参数"]
["Swift", "泛型", "类型参数", "类型参数名"]
```

扩展泛型类型

当你扩展一个泛型类型的时候（使用 extension 关键字），你并不需要在扩展的定义中提供类型参数列表。更加方便的是，原始类型定义中声明的类型参数列表在扩展里是可以使用的，并且这些来自原始类型中的参数名称会被用作原始定义中类型参数的引用。

实例

```
struct TOS<T> {
    var items = [T]()
    mutating func push(item: T) {
        items.append(item)
    }

    mutating func pop() -> T {
        return items.removeLast()
    }
}

var tos = TOS<String>()
tos.push("Swift")
print(tos.items)

tos.push("泛型")
    print(tos.items)

    tos.push("类型参数")
    print(tos.items)

    tos.push("类型参数名")
    print(tos.items)

// 扩展泛型 TOS 类型
extension TOS {
    var first: T? {
        return items.isEmpty ? nil : items[items.count - 1]
    }
}

if let first = tos.first {
    print("栈顶部项：\(first)")
}
```

以上程序执行输出结果为：

```
["Swift"]
["Swift", "泛型"]
["Swift", "泛型", "类型参数"]
["Swift", "泛型", "类型参数", "类型参数名"]
栈顶部项：类型参数名
```

类型约束

类型约束指定了一个必须继承自指定类的类型参数，或者遵循一个特定的协议或协议构成。

类型约束语法

你可以写一个在一个类型参数名后面的类型约束，通过冒号分割，来作为类型参数链的一部分。这种作用于泛型函数的类型约束的基础语法如下所示（和泛型类型的语法相同）：

```
func someFunction<T: SomeClass, U: SomeProtocol>(someT: T, someU: U
    // 这里是函数主体
}
```

实例

```
// 函数可以作用于查找一字符串数组中的某个字符串
func findStringIndex(array: [String], _ valueToFind: String) -> Int
    for (index, value) in array.enumerate() {
        if value == valueToFind {
            return index
        }
    }
    return nil
}

let strings = ["cat", "dog", "llama", "parakeet", "terrapin"]
if let foundIndex = findStringIndex(strings, "llama") {
    print("llama 的下标索引值为 \(foundIndex)")
}
```

以上程序执行输出结果为：

```
llama 的下标索引值为 2
```

关联类型实例

Swift 中使用 typealias 关键字来设置关联类型。

定义一个协议时，有的时候声明一个或多个关联类型作为协议定义的一部分是非常有用的。

```
protocol Container {
    // 定义了一个ItemType关联类型
    typealias ItemType
    mutating func append(item: ItemType)
    var count: Int { get }
    subscript(i: Int) -> ItemType { get }
}

// 遵循Container协议的泛型TOS类型
struct TOS<T>: Container {
    // original Stack<T> implementation
    var items = [T]()
    mutating func push(item: T) {
        items.append(item)
    }

    mutating func pop() -> T {
        return items.removeLast()
    }

    // conformance to the Container protocol
    mutating func append(item: T) {
        self.push(item)
    }

    var count: Int {
        return items.count
    }

    subscript(i: Int) -> T {
        return items[i]
    }
}

var tos = TOS<String>()
tos.push("Swift")
print(tos.items)

tos.push("泛型")
print(tos.items)

tos.push("参数类型")
print(tos.items)

tos.push("类型参数名")
print(tos.items)
```

以上程序执行输出结果为：

```
["Swift"]
["Swift", "泛型"]
["Swift", "泛型", "参数类型"]
["Swift", "泛型", "参数类型", "类型参数名"]
```

Where 语句

类型约束能够确保类型符合泛型函数或类的定义约束。

你可以在参数列表中通过where语句定义参数的约束。

你可以写一个where语句，紧跟在在类型参数列表后面，where语句后跟一个或者多个针对关联类型的约束，以及（或）一个或多个类型和关联类型间的等价(equality)关系。

实例

下面的例子定义了一个名为allItemsMatch的泛型函数，用来检查两个Container实例是否包含相同顺序的相同元素。

如果所有的元素能够匹配，那么返回一个为true的Boolean值，反之则为false。

```
protocol Container {
    typealias ItemType
    mutating func append(item: ItemType)
    var count: Int { get }
    subscript(i: Int) -> ItemType { get }
}

struct Stack<T>: Container {
    // original Stack<T> implementation
    var items = [T]()
    mutating func push(item: T) {
        items.append(item)
    }

    mutating func pop() -> T {
        return items.removeLast()
    }

    // conformance to the Container protocol
    mutating func append(item: T) {
        self.push(item)
    }

    var count: Int {
        return items.count
    }
```

```
    subscript(i: Int) -> T {
        return items[i]
    }
}

func allItemsMatch<
    C1: Container, C2: Container
    where C1.ItemType == C2.ItemType, C1.ItemType: Equatable>
    (someContainer: C1, anotherContainer: C2) -> Bool {
        // 检查两个Container的元素个数是否相同
        if someContainer.count != anotherContainer.count {
            return false
        }

        // 检查两个Container相应位置的元素彼此是否相等
        for i in 0..<someContainer.count {
            if someContainer[i] != anotherContainer[i] {
                return false
            }
        }
        // 匹配所有项，返回 true
        return true
}

var tos = Stack<String>()
tos.push("Swift")
print(tos.items)

tos.push("泛型")
print(tos.items)

tos.push("Where 语句")
print(tos.items)

var eos = ["Swift", "泛型", "Where 语句"]
print(eos)
```

以上程序执行输出结果为：

```
["Swift"]
["Swift", "泛型"]
["Swift", "泛型", "Where 语句"]
["Swift", "泛型", "Where 语句"]
```

Swift 访问控制

访问控制可以限定其他源文件或模块中代码对你代码的访问级别。

你可以明确地给单个类型（类、结构体、枚举）设置访问级别，也可以给这些类型的属性、函数、初始化方法、基本类型、下标索引等设置访问级别。

协议也可以被限定在一定的范围内使用，包括协议里的全局常量、变量和函数。

访问控制基于模块与源文件。

模块指的是以独立单元构建和发布的Framework或Application。在Swift 中的一个模块可以使用import关键字引入另外一个模块。

源文件是单个源码文件，它通常属于一个模块， 源文件可以包含多个类和函数 的定义。

Swift 为代码中的实体提供了三种不同的访问级别:public、internal、private。

访问级别	定义
Public	可以访问自己模块中源文件里的任何实体，别人也可以通过引入该模块来访问源文件里的所有实体。
Internal	：可以访问自己模块中源文件里的任何实体，但是别人不能访问该模块中源文件里的实体。
Private	只能在当前源文件中使用的实体，称为私有实体。

public为最高级访问级别，private为最低级访问级别。

语法

通过修饰符public、internal、private来声明实体的访问级别：

```
public class SomePublicClass {}
internal class SomeInternalClass {}
private class SomePrivateClass {}

public var somePublicVariable = 0
internal let someInternalConstant = 0
private func somePrivateFunction() {}
```

除非有特殊的说明，否则实体都使用默认的访问级别internal。

函数类型访问权限

函数的访问级别需要根据该函数的参数类型和返回类型的访问级别得出。

下面的例子定义了一个名为someFunction全局函数，并且没有明确地申明其访问级别。

```
func someFunction() -> (SomeInternalClass, SomePrivateClass) {
    // 函数实现
}
```

函数中其中一个类 SomeInternalClass 的访问级别是internal，另一个 SomePrivateClass 的访问级别是private。所以根据元组访问级别的原则，该元组的访问级别是private。

```
private func someFunction() -> (SomeInternalClass, SomePrivateClass
    // 函数实现
}
```

将该函数申明为public或internal，或者使用默认的访问级别internal都是错误的。

枚举类型访问权限

枚举中成员的访问级别继承自该枚举，你不能为枚举中的成员单独申明不同的访问级别。

实例

比如下面的例子，枚举 Student 被明确的申明为 public 级别，那么它的成员 Name，Mark 的访问级别同样也是 public：

```
public enum Student {
    case Name(String)
    case Mark(Int,Int,Int)
}

var studDetails = Student.Name("Swift")
var studMarks = Student.Mark(98,97,95)

switch studMarks {
case .Name(let studName):
    print("学生名: \(studName).")
case .Mark(let Mark1, let Mark2, let Mark3):
    print("学生成绩: \(Mark1),\(Mark2),\(Mark3)")
}
```

以上程序执行输出结果为：

```
学生成绩: 98,97,95
```

子类访问权限

子类的访问级别不得高于父类的访问级别。比如说，父类的访问级别是internal，子类的访问级别就不能申明为public。

```
public class SuperClass {
    private func show() {
        print("超类")
    }
}

// 访问级别不能低于超类 internal > public
internal class SubClass: SuperClass  {
    override internal func show() {
        print("子类")
    }
}

let sup = SuperClass()
sup.show()

let sub = SubClass()
sub.show()
```

以上程序执行输出结果为：

```
超类
子类
```

常量、变量、属性、下标访问权限

常量、变量、属性不能拥有比它们的类型更高的访问级别。

比如说，你定义一个public级别的属性，但是它的类型是private级别的，这是编译器所不允许的。

同样，下标也不能拥有比索引类型或返回类型更高的访问级别。

如果常量、变量、属性、下标索引的定义类型是private级别的，那么它们必须要明确的申明访问级别为private:

```
private var privateInstance = SomePrivateClass()
```

Getter 和 Setter访问权限

常量、变量、属性、下标索引的Getters和Setters的访问级别继承自它们所属成员的访问级别。

Setter的访问级别可以低于对应的Getter的访问级别，这样就可以控制变量、属性或下标索引的读写权限。

```
class Samplepgm {
    private var counter: Int = 0{
        willSet(newTotal){
            print("计数器: \(newTotal)")
        }
        didSet{
            if counter > oldValue {
                print("新增加数量 \(counter - oldValue)")
            }
        }
    }
}

let NewCounter = Samplepgm()
NewCounter.counter = 100
NewCounter.counter = 800
```

以上程序执行输出结果为：

```
计数器: 100
新增加数量 100
计数器: 800
新增加数量 700
```

构造器和默认构造器访问权限

初始化

我们可以给自定义的初始化方法申明访问级别，但是要不高于它所属类的访问级别。但必要构造器例外，它的访问级别必须和所属类的访问级别相同。

如同函数或方法参数，初始化方法参数的访问级别也不能低于初始化方法的访问级别。

默认初始化方法

Swift为结构体、类都提供了一个默认的无参初始化方法，用于给它们的所有属性提供赋值操作，但不会给出具体值。

默认初始化方法的访问级别与所属类型的访问级别相同。

实例

在每个子类的 init() 方法前使用 required 关键字声明访问权限。

```
class classA {
    required init() {
        var a = 10
        print(a)
    }
}

class classB: classA {
    required init() {
        var b = 30
        print(b)
    }
}

let res = classA()
let show = classB()
```

以上程序执行输出结果为：

```
10
30
10
```

协议访问权限

如果想为一个协议明确的申明访问级别，那么需要注意一点，就是你要确保该协议只在你申明的访问级别作用域中使用。

如果你定义了一个public访问级别的协议，那么实现该协议提供的必要函数也会是public的访问级别。这一点不同于其他类型，比如，public访问级别的其他类型，他们成员的访问级别为internal。

```
public protocol TcpProtocol {
    init(no1: Int)
}

public class MainClass {
    var no1: Int // local storage
    init(no1: Int) {
        self.no1 = no1 // initialization
    }
}

class SubClass: MainClass, TcpProtocol {
    var no2: Int
    init(no1: Int, no2 : Int) {
        self.no2 = no2
        super.init(no1:no1)
    }

    // Requires only one parameter for convenient method
    required override convenience init(no1: Int)  {
        self.init(no1:no1, no2:0)
    }
}

let res = MainClass(no1: 20)
let show = SubClass(no1: 30, no2: 50)

print("res is: \(res.no1)")
print("res is: \(show.no1)")
print("res is: \(show.no2)")
```

以上程序执行输出结果为：

```
res is: 20
res is: 30
res is: 50
```

扩展访问权限

你可以在条件允许的情况下对类、结构体、枚举进行扩展。扩展成员应该具有和原始类成员一致的访问级别。比如你扩展了一个公共类型，那么你新加的成员应该具有和原始成员一样的默认的internal访问级别。

或者，你可以明确申明扩展的访问级别（比如使用private extension）给该扩展内所有成员申明一个新的默认访问级别。这个新的默认访问级别仍然可以被单独成员所申明的访问级别所覆盖。

泛型访问权限

泛型类型或泛型函数的访问级别取泛型类型、函数本身、泛型类型参数三者中的最低访问级别。

```
public struct TOS<T> {
    var items = [T]()
    private mutating func push(item: T) {
        items.append(item)
    }

    mutating func pop() -> T {
        return items.removeLast()
    }
}

var tos = TOS<String>()
tos.push("Swift")
print(tos.items)

tos.push("泛型")
print(tos.items)

tos.push("类型参数")
print(tos.items)

tos.push("类型参数名")
print(tos.items)
let deletetos = tos.pop()
```

以上程序执行输出结果为：

```
["Swift"]
["Swift", "泛型"]
["Swift", "泛型", "类型参数"]
["Swift", "泛型", "类型参数", "类型参数名"]
```

类型别名

任何你定义的类型别名都会被当作不同的类型，以便于进行访问控制。一个类型别名的访问级别不可高于原类型的访问级别。

比如说，一个private级别的类型别名可以设定给一个public、internal、private的类型，但是一个public级别的类型别名只能设定给一个public级别的类型，不能设定给internal或private 级别的类型。

> 注意： 这条规则也适用于为满足协议一致性而给相关类型命名别名的情况。

```
public protocol Container {
    typealias ItemType
    mutating func append(item: ItemType)
    var count: Int { get }
    subscript(i: Int) -> ItemType { get }
}

struct Stack<T>: Container {
    // original Stack<T> implementation
    var items = [T]()
    mutating func push(item: T) {
        items.append(item)
    }

    mutating func pop() -> T {
        return items.removeLast()
    }

    // conformance to the Container protocol
    mutating func append(item: T) {
        self.push(item)
    }

    var count: Int {
        return items.count
    }

    subscript(i: Int) -> T {
        return items[i]
    }
}

func allItemsMatch<
```

```
    C1: Container, C2: Container
    where C1.ItemType == C2.ItemType, C1.ItemType: Equatable>
    (someContainer: C1, anotherContainer: C2) -> Bool {
        // check that both containers contain the same number of it
        if someContainer.count != anotherContainer.count {
            return false
        }

        // check each pair of items to see if they are equivalent
        for i in 0..<someContainer.count {
            if someContainer[i] != anotherContainer[i] {
                return false
            }
        }

        // all items match, so return true
        return true
}

var tos = Stack<String>()
tos.push("Swift")
print(tos.items)

tos.push("泛型")
print(tos.items)

tos.push("Where 语句")
print(tos.items)

var eos = ["Swift", "泛型", "Where 语句"]
print(eos)
```

以上程序执行输出结果为：

```
["Swift"]
["Swift", "泛型"]
["Swift", "泛型", "Where 语句"]
["Swift", "泛型", "Where 语句"]
```

W3School ios教程

来源：ios教程

整理：飞龙

IOS 简介

IOS之前被称为iPhone OS，是一个由苹果公司开发的移动操作系统。

iOS的第一个版本是在2007年发布的，其中包括iPhone和iPod Touch。

2004年4月发布iPad（第一代），并于2012年11月发布了iPad迷你款。

IOS设备发布相当频繁，由以往经验可知，每年都会推出至少一个版本的iPhone和iPad。

现在发布了iPhone5s，之前还推出了iPhone，iPhone3gs，iPhone4,iPhone4s以及iphone5。

同样的iPad也从iPad一代更新到iPad四代以及一个特别的迷你版iPad。

iOS SDK已经从1.0更新到6.0。最新的iOS SDK6.0，是唯一支持Xcode4.5和其更高版本的版本。

丰富的苹果文档，使我们能找到许多方法和库用于我们的部署目标。在Xcode的当前版本中，我们能够在iOS4.3,5.0和6.0的部署目标之间选择。

IOS的影响能够从以下的特点显现:

Facebook和Twitter上，加速度计，GPS，高端处理器，相机，Safari浏览器，功能强大的API，游戏中心，在应用程序内购买，提醒，宽范围的手势

- 地图
- Siri
- Facebook 和 Twitter
- Multi-Touch（多点触摸）
- Accelerometer（加速度传感器）
- GPS
- 高性能处理器
- 相机
- Safari浏览器
- 功能强大的API
- 游戏中心
- 在应用程序内购买
- 提醒功能
- 手势

iPhone和iPad的用户日益增多，这为iPhone和iPad应用商城的研发者创造了赚钱的机遇。

IOS最新的一点是，苹果公司研发了应用商城，这样用户可以购买应用程序来完善他们的iOS设备。

研发者可以在应用商城发布免费和付费的应用软件。

开发应用程序并将其发布到应用商店，开发人员需要注册iOS开发者计划，为其发展更新Xcode每年话费99美元和Mac Mountain Lion 或更高。

注册Apple开发者

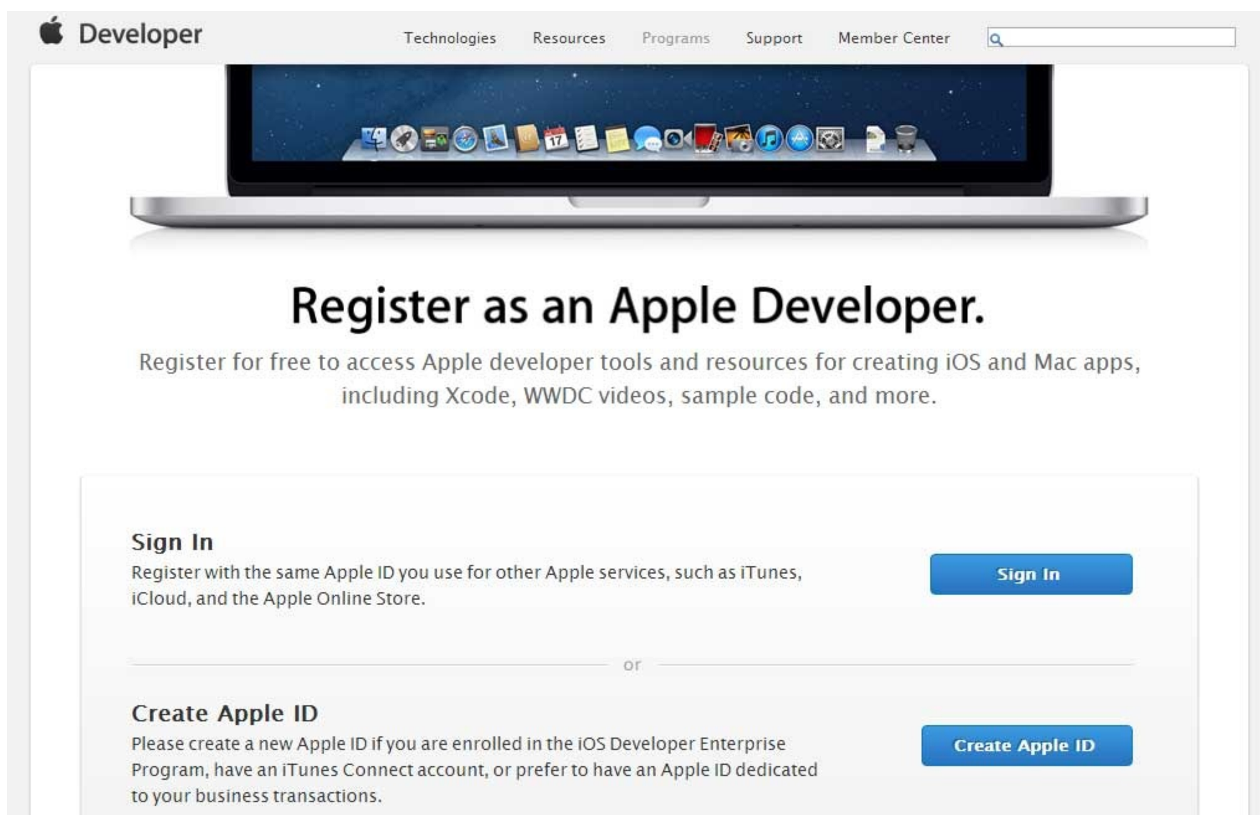
对拥有Apple设备的用户来说，非常有必要拥有Apple ID，而且成为一个研发者，必须用到Apple ID,获取 Apple ID是免费的，也无需有资费方面的顾虑。

拥有Apple账户有以下好处：

- 易于了解研发工具；
- 全球研发者视频会议；
- 受邀加入iOS研发者团队;

注册苹果账号

1、单击 (https://developer.apple.com/programs/register/) 并选择创建Apple ID



2、输入个人信息

3、返回邮箱确认，激活账号

4、下载研发工具，Xcode及它所包含的iOS模拟器，iOS SDK和其他研发资源

申请APP开发者

1、点击 (https://developer.apple.com/programs/ios/)



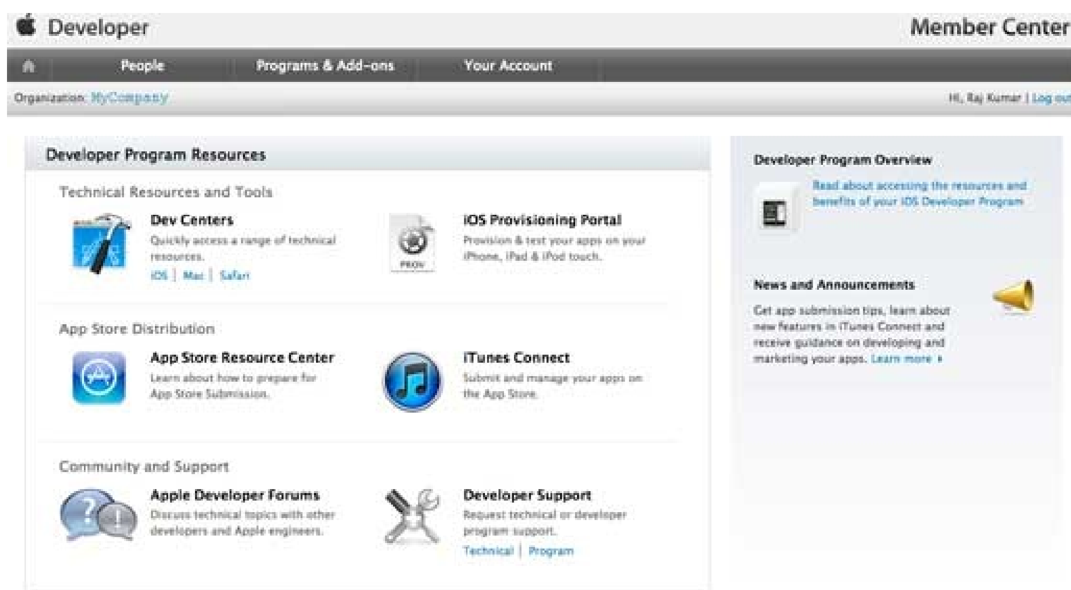
2、点击注册页面

3、登录账号（已有账号）或注册Apple ID

4、选择个人账号或公司账号，研发者团队使用公司账号，个人账号不能添加其他用户

5、新用户进入个人信息页面，使用信用卡购买加入研发项目

6、选择会员中心，利用研发者资源



7、在此处可以执行以下操作:

- 创建资源调配的配置文件
- 管理团队和设备
- 通过iTunes Connect管理应用到应用程序
- 获取论坛和技术支持

IOS Xcode 安装

1、从 https://developer.apple.com/downloads/ 下载Xcode的最新版本。



2、双击Xcode dmg文件

3、将找到的设备安装和打开

4、在这里会有两个项目在显示的窗口中即Xcode应用程序和应用程序文件夹的快捷方式

5、将Xcode拖拽并复制到应用程序

6、在应用里选择和运行程序，Xcode也将成为运行程序中的一部分

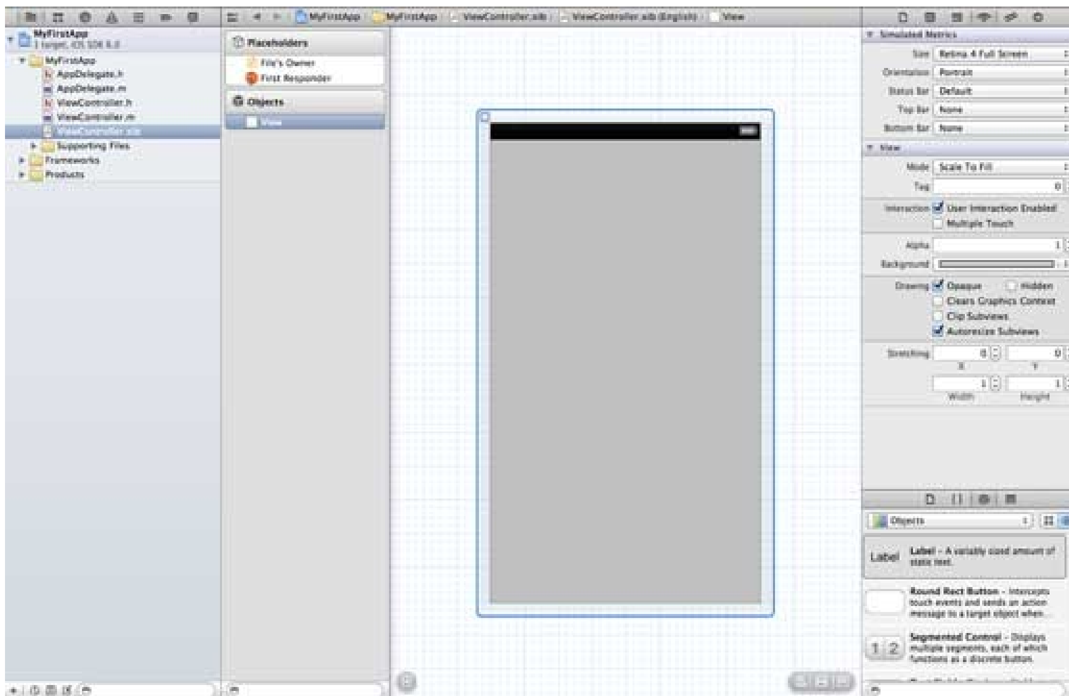
还可以从Mac App store里下载Xcode，并按照屏幕上的安装步

界面生成器（Interface Builder）

利用界面生成器这一工具，能很容易的创建UI界面。

可利用一系列的UI元素，拖拽进入UI可视界面。

我们将在接下来的页面了解添加用户界面元素，创建零售商和UI元素的操作。



在对象库的下方包含有全部必要的UI元素。用户界面通常称为xibs，这是他们的文件扩展名。

每个xibs都链接到相应的试图控制器。

IOS模拟器

IOS模拟器实际上包含两种类型的设备即iPhone和iPad及其不同的版本。

iPhone版本包括iPhone（常规版）、iPhone Retina，iPhone5,iPhone53。

Ipad有iPad和iPad Retina。iPhone模拟器显示如下:



你可以在经度和纬度影响应用程序的位置的情况下运行iOS模拟器，也可以模拟内存警告和呼叫在模拟器中的状态。

能够多数目的使用模拟器，但不能测试像加速度计这样的设备的功能。因此你可能需要iOS设备来测试一个应用程序的所有方面。

Objective-C 简介

在iOS的开发中使用的是Objective C语言，它是一中面向对象的语言，因而对于已经掌握面向对象语言知识的编程者来说是非常简单的。

接口和实现

在Objective里完成的文件被称为界面文件，该类文件的定义被称为实现文件。

一个简单的界面文件MyClass.h将如图所示:

```
@interace MyClass:NSObject{
// class variable declared here
}
// class properties declared here
// class methods and instance methods declared here
@end
```

执行MyClass.m文件，如下所示

```
@implementation MyClass
// class methods defined here
@end
```

创建对象

完成创建对象，如下所示

```
MyClass  *objectName = [[MyClass alloc]init] ;
```

方法（methods）

Objective C中声明的方法如下所示:

```
-(returnType)methodName:(typeName) variable1 :(typeName)variable2;
```

下面显示了一个示例:

```
-(void)calculateAreaForRectangleWithLength:(CGfloat)length
andBreadth:(CGfloat)breadth;
```

你可能会想什么是andBreadth字符串，其实它的可选字符串可以帮助我们阅读和理解方法，尤其是当方法被调用的时候。

在同一类中调用此方法，我们使用下面的语句。

```
[self calculateAreaForRectangleWithLength:30 andBreadth:20];
```

正如上文所说的andBreath使用有助于我们理解breath是20。Self用来指定它是一个类的方法。

类方法（**class methods**）

直接而无需创建的对象，可以访问类方法。他们没有任何变量和它关联的对象。示例如下:

```
+(void)simpleClassMethod;
```

它可以通过使用类名（假设作为MyClass类名称）访问，如下所示:

```
[MyClass simpleClassMethod];
```

实例方法

可以创建的类的对象后只访问实例方法，内存分配到的实例变量。实例方法如下所示:

```
-(void)simpleInstanceMethod;
```

创建类的对象后，它可以访问它。如下所示:

```
MyClass  *objectName = [[MyClass alloc]init] ;
[objectName simpleInstanceMethod];
```

Objective C的重要数据类型

数据类型
NSString 字符串
CGfloat 浮点值的基本类型
NSInteger 整型
BOOL 布尔型

打印日志

NSLog用于打印一份声明，它将打印在设备日志和调试版本的控制台和分别调试模式上。

如 NSlog(@"");

控制结构

除了几个增补的条款外，大多数的控制结构与C以及C++相同

属性（**properties**）

用于访问类的外部类的变量属性

比如：@property（非原子、强）NSString*myString

访问属性

可以使用点运算符访问属性，若要访问上一属性可以执行以下操作

```
self.myString = @"Test";
```

还可以使用set的方法，如下所示：

```
[self setMyString:@"Test"];
```

类别（**categories**）

类用于将方法添加到现有类。通过这种方法可以将方法添加到类，甚至不用执行文件，就可以在其中定义实际的类。MyClass的样本类别，如下所示：

```
@interace MyClass(customAdditions)
- (void)sampleCategoryMethod;
@end

@implementation MyClass(categoryAdditions)

-(void)sampleCategoryMethod{
   NSLog(@"Just a test category");
}
```

数组（arrays）

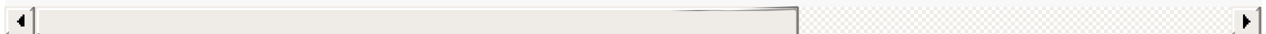
NSMutable和NSArray 是ObjectiveC中使用的数组类，前者是可变数组，后者是不可变数组。如下:

```
NSMutableArray *aMutableArray = [[NSMutableArray alloc]init];
[anArray addObject:@"firstobject"];
NSArray *aImmutableArray = [[NSArray alloc]
initWithObjects:@"firstObject",nil];
```

词典

NSMutableDictionary和NSDictionary是Objective中使用的字典，前者可变词典，后者不可变词典，如下所示:

```
NSMutableDictionary*aMutableDictionary = [[NSMutableArray alloc]in
[aMutableDictionary setObject:@"firstobject" forKey:@"aKey"];
NSDictionary*aImmutableDictionary= [[NSDictionary alloc]initWithOb
@"firstObject",nil] forKeys:[ NSArray arrayWithObjects:@"aKey"]];
```



创建第一款iPhone应用程序

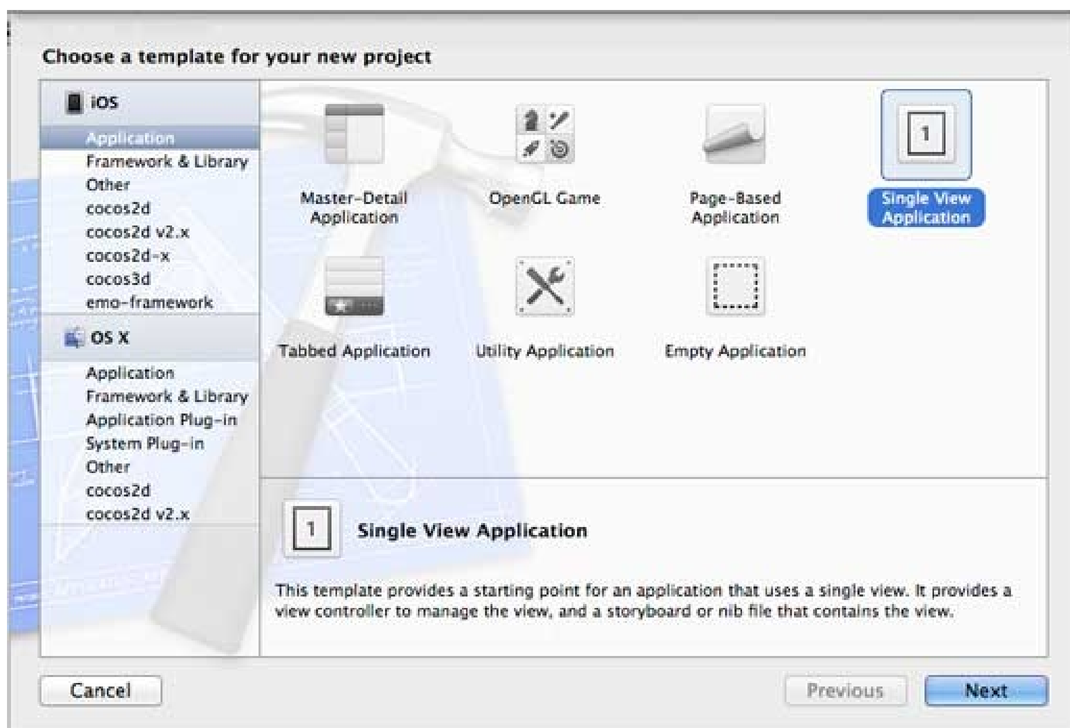
现在让我们来创建一个在iOS模拟器上运行的简单视图应用（空白的应用程序）。

操作步骤如下：

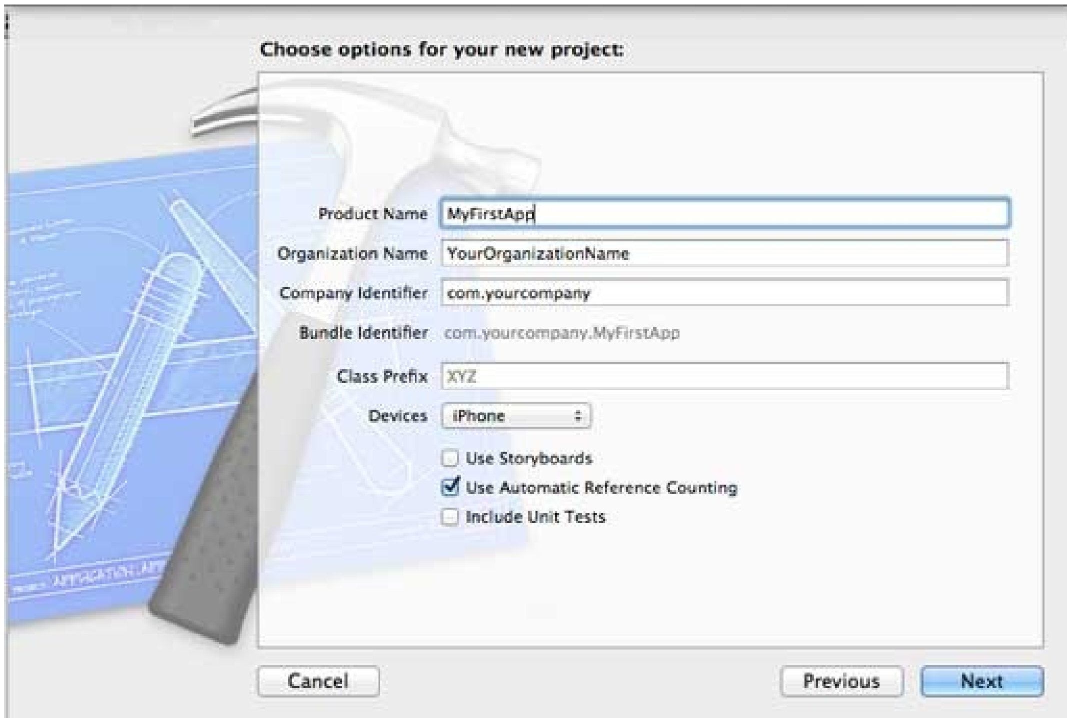
1、打开Xcode并选择创建一个新的Xcode项目。



2. 然后选择单一视图应用程序

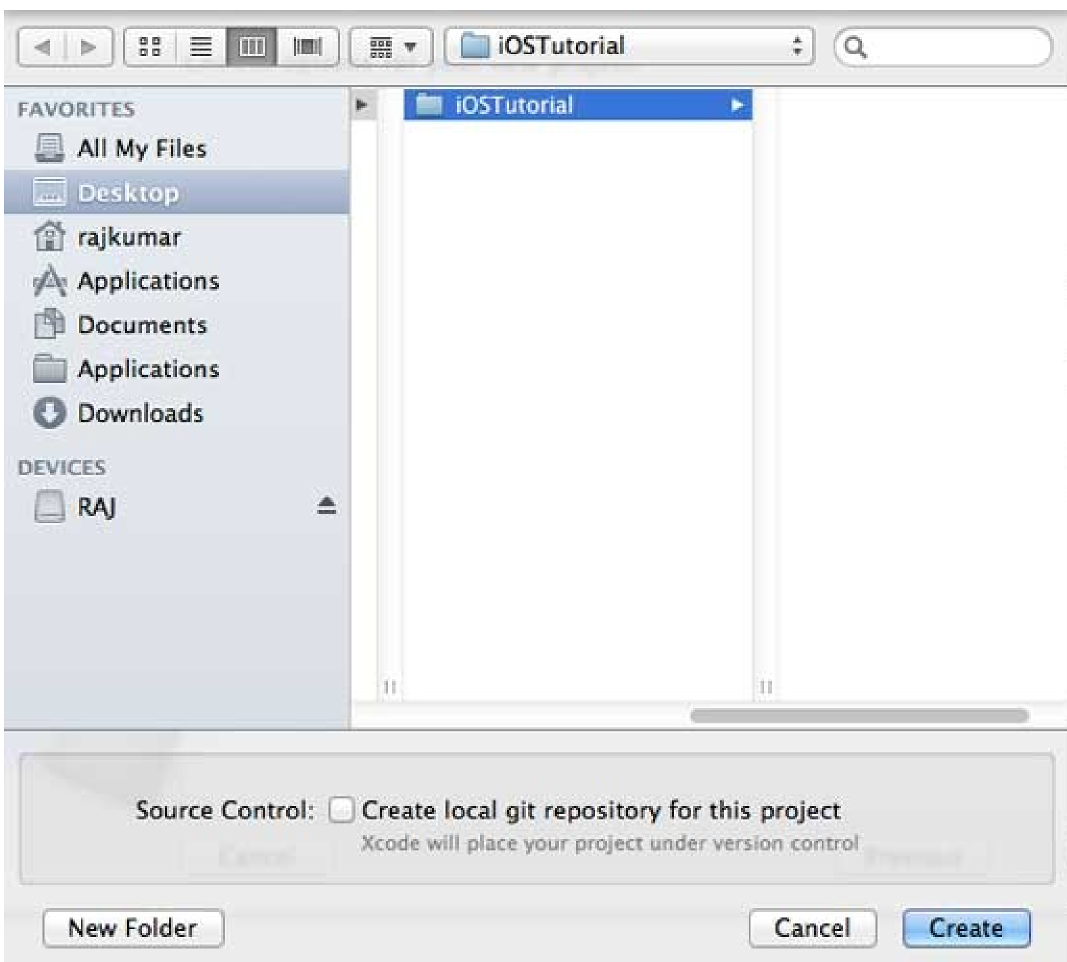


3. 接下来输入产品名称即应用程序名称、组织名称和公司标识符。

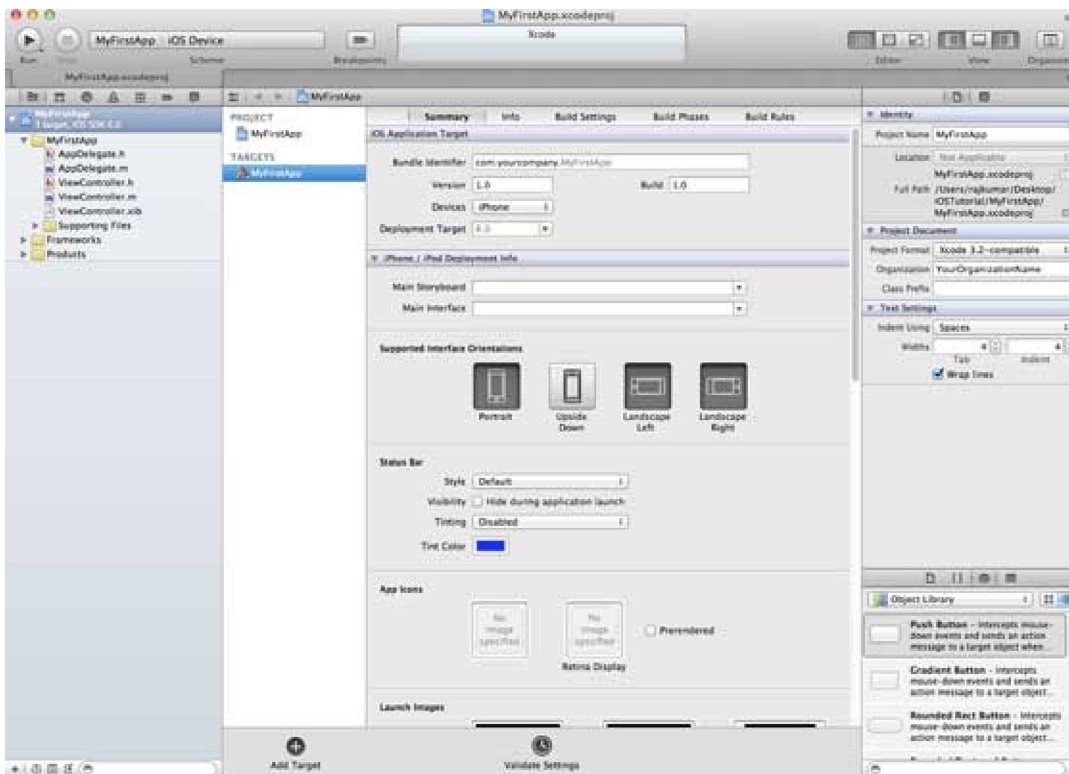


4. 确定已经选择自动应用计数，以自动释放超出范围的资源。单击下一步。

5.选择项目目录并选择创建

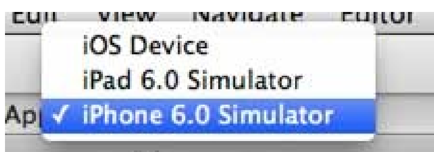


6. 你将看到如下所示的页面

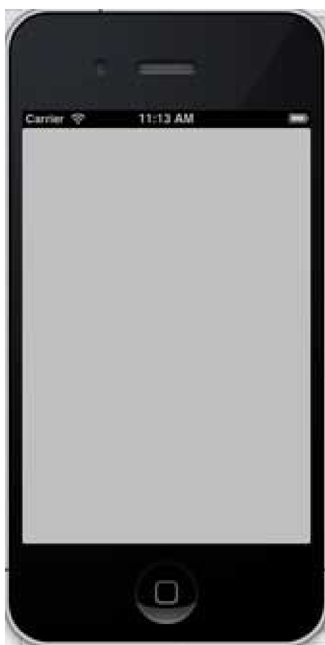


屏幕上方能够设置方向、生成和释放。有一个部署目标，设备支持4.3及以上版本的部署目标，这些不是必须的，现在只要专注于运行该应用程序。

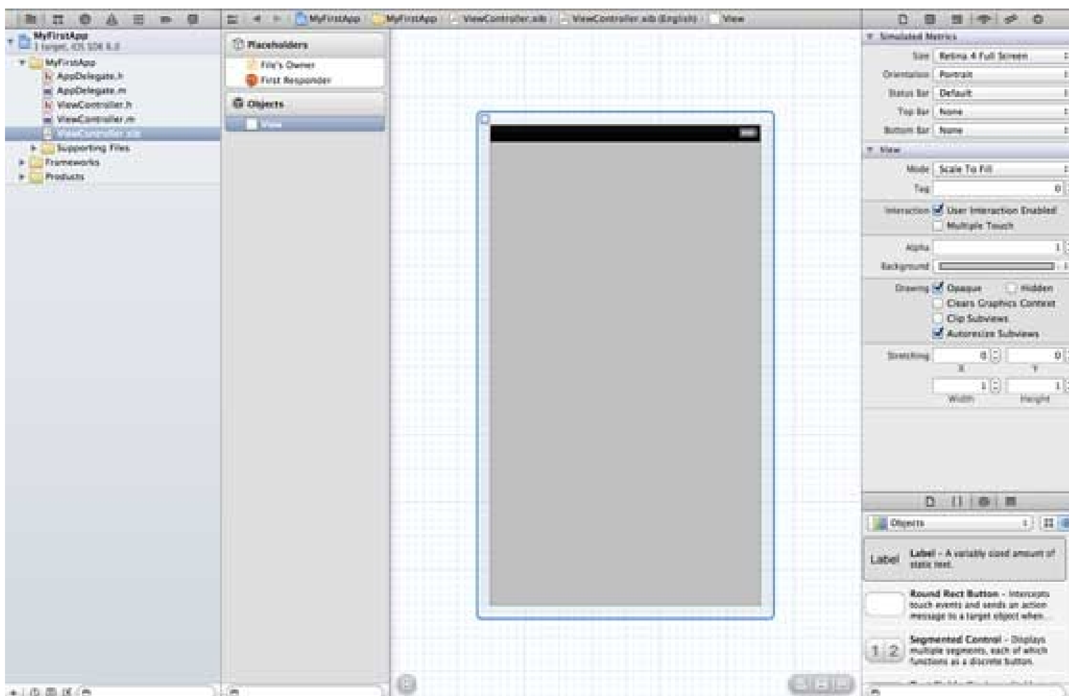
7. 在下拉菜单中选择iPhone Simulator并运行。



8. 成功运行第一个应用程序，将得到的输出，如下所示。



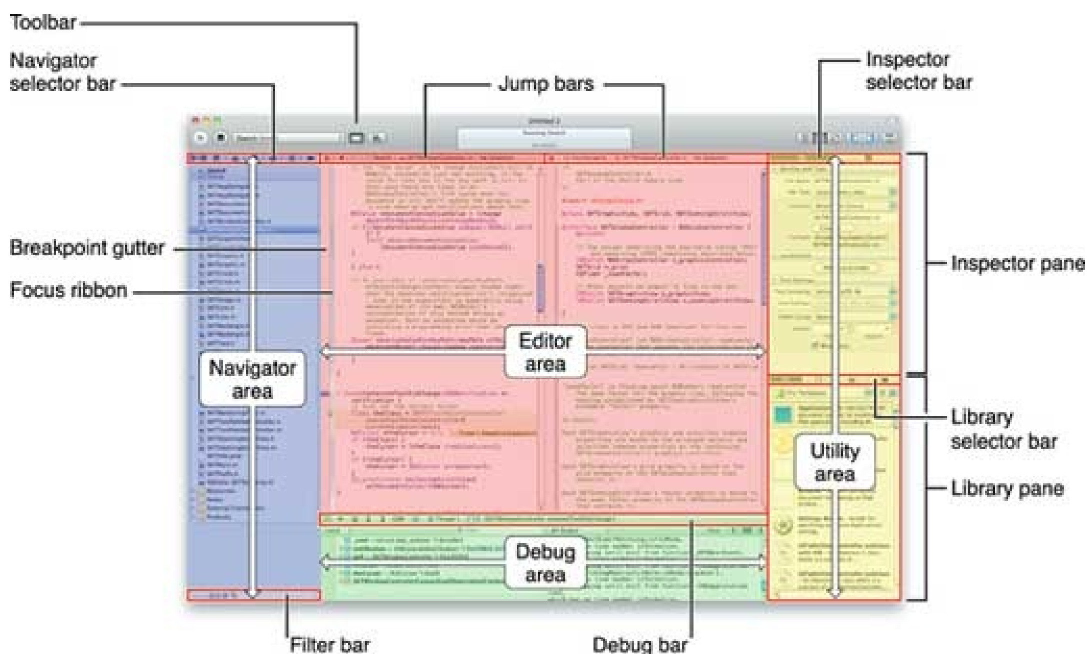
更改背景颜色使之有开始的界面生成器。选择ViewController.xib。在右侧选择背景选项，更改颜色并运行。



在上述项目中，默认情况下，部署目标已设置为iOS6.0且自动布局将被启用。

为确保应用程序能iOS4.3设备上正常运行，我们已经在开始创建应用程序时修改了部署目标，但我们不禁用自动布局，要取消自动布局，我们需要取消选择自动班上复选框在文件查看器的每个nib，也就是xib文件。

Xcode项目IDE的各部分显示如下（苹果Xcode4用户文档）



在上面所示的检查器选择器栏中可以找到文件检查器，且可以取消选择自动布局。当你想要的目标只有iOS6.0的设备时，可以使用自动布局。

当然，也可以使用新功能，如当加注到iOS6时，就可以使用passbook这一功能。现在，以Ios4.3作为部署目标。

深入了解第一款IOS应用程序代码

5个不同文件生成应用程序，如下所示

- AppDelegate.h
- AppDelegate.m
- ViewController.h
- ViewController.m
- ViewController.xib

我们使用单行注释（//）来解释简单代码，重要的项目代码解释在代码下方。

AppDelegate.h

```
// Header File that provides all UI related items.
#import &lt;UIKit/UIKit.h&gt;
 // Forward declaration (Used when class will be defined /imported
@class ViewController;

 // Interface for Appdelegate
@interface AppDelegate : UIResponder &lt;UIApplicationDelegate&gt;
// Property window
@property (strong, nonatomic) UIWindow *window;
 // Property Viewcontroller
@property (strong, nonatomic) ViewController *viewController;
//this marks end of interface
@end
```

代码说明

- AppDelegate调用UIResponder来处理Ios事件。
- 完成UIApplication的命令，提供关键应用程序事件，如启动完毕，终止，等等
- 在iOS设备的屏幕上用UIWindow对象来管理和协调各种视角，它就像其它加载视图的基本视图一样。通常一个应用程序只有一个窗口。
- UIViewController来处理屏幕流

AppDelegate.m

```
// Imports the class Appdelegate's interface
import "AppDelegate.h"

// Imports the viewcontroller to be loaded
#import "ViewController.h"
```

```
// Class definition starts here
@implementation AppDelegate

// Following method intimates us the application launched  successf
- (BOOL)application:(UIApplication *)application
 didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    self.window = [[UIWindow alloc] initWithFrame:
    [[UIScreen mainScreen] bounds]];
    // Override point for customization after application launch.
    self.viewController = [[ViewController alloc]
     initWithNibName:@"ViewController" bundle:nil];
    self.window.rootViewController = self.viewController;
    [self.window makeKeyAndVisible];
    return YES;
}

- (void)applicationWillResignActive:(UIApplication *)application
{
    /* Sent when the application is about to move from active to in
    This can occur for certain types of temporary interruptions
    (such as an incoming phone call or SMS message)
    or when the user quits the application and it begins the transi
    background state. Use this method to pause ongoing tasks, disab
    and throttle down OpenGL ES frame rates. Games should use this
    to pause the game.*/
}

- (void)applicationDidEnterBackground:(UIApplication *)application
{
    /* Use this method to release shared resources, save user data,
    timers, and store enough application state information    to re
    application to its current state in case it is terminated later
    application supports background execution, this method is calle
    of applicationWillTerminate: when the user quits.*/
}

- (void)applicationWillEnterForeground:(UIApplication *)applicatior
{
    /* Called as part of the transition from the background to the
    here you can undo many of the changes made on entering the back
}

- (void)applicationDidBecomeActive:(UIApplication *)application
{
    /* Restart any tasks that were paused (or not yet started) whil
    application was inactive. If the application was previously in
    optionally refresh the user interface.*/
}

- (void)applicationWillTerminate:(UIApplication *)application
{
```

```
    /* Called when the application is about to terminate. Save data
    See also applicationDidEnterBackground:. */
}

@end
```

代码说明

- 此处定义UIApplication。上面定义的所有方法都是应用程序UI调动和不包含任何用户定义的方法。
- UIWindow对象被分配用来保存应用程序分配对象。
- UIController作为窗口初始视图控制器
- 调用makeKeyAndVisible能使窗口可见

ViewController.h

```
#import

// Interface for class ViewController
@interface ViewController : UIViewController

@end
```

代码说明

- ViewController类继承UIViewController，为iOS应用程序提供基本视图管理模型。

ViewController.m

```
#import "ViewController.h"

// Category, an extension of ViewController class
@interface ViewController ()

@end

@implementation ViewController

- (void)viewDidLoad
{
    [super viewDidLoad];
    // Do any additional setup after loading the view, typically fr
}

- (void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}

@end
```

代码说明

- 在这里两种方法实现UIViewController类的基类中定义
- 初始视图加载后调用viewDidLoad中的安装程序
- 在内存警告的情况下调用didReceviveMemoryWarning

简介

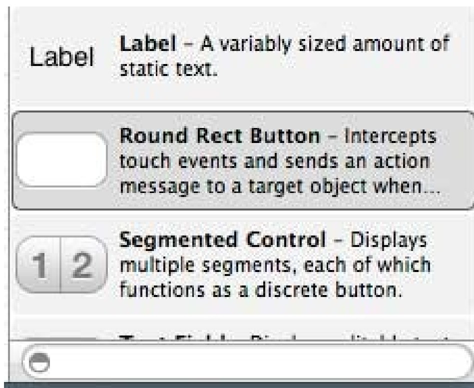
在iOS中，操作（action）和输出口（Outlet）指的是ibActions和ibOutlets，也就是ib接口生成器所在的地方。这些都和UI元素相关，我们将直观的了解他们后探讨如何实现他们。

步骤

1、让我们使用第一款iPhone应用程序。

2、从导航部分中的文件中选择ViewController.xib文件

3、从右手边得窗口下面显示的窗口格库中选择UI元素



4、拖拽UI元素到界面生成器的可视框中

5、添加标签和红色圆形按钮到可视图中



6、在工作区工具栏的右上角找到编辑器选择按钮，如下图所示

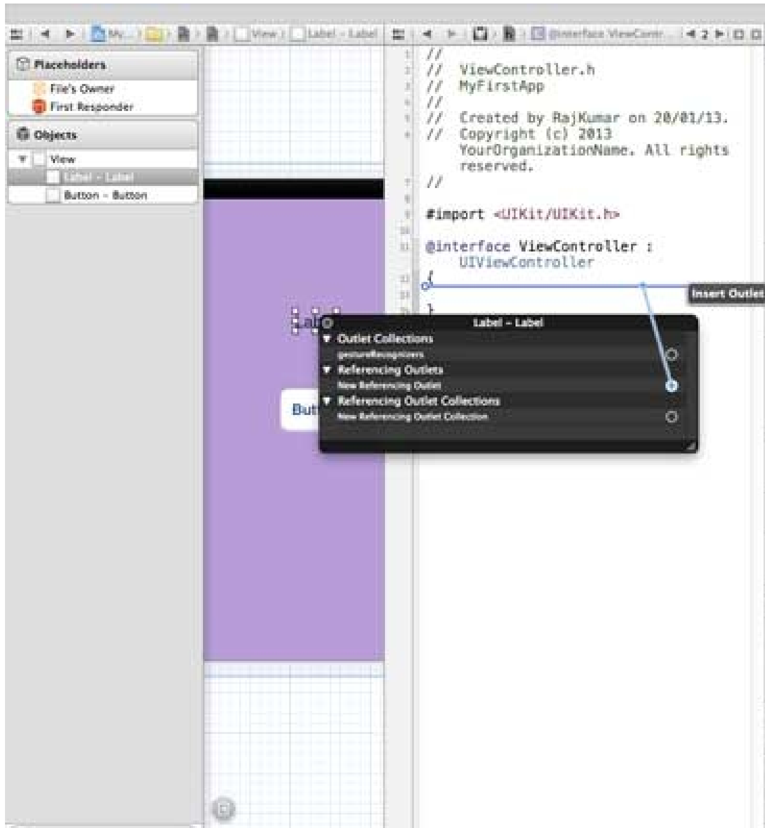


选择编辑器按钮

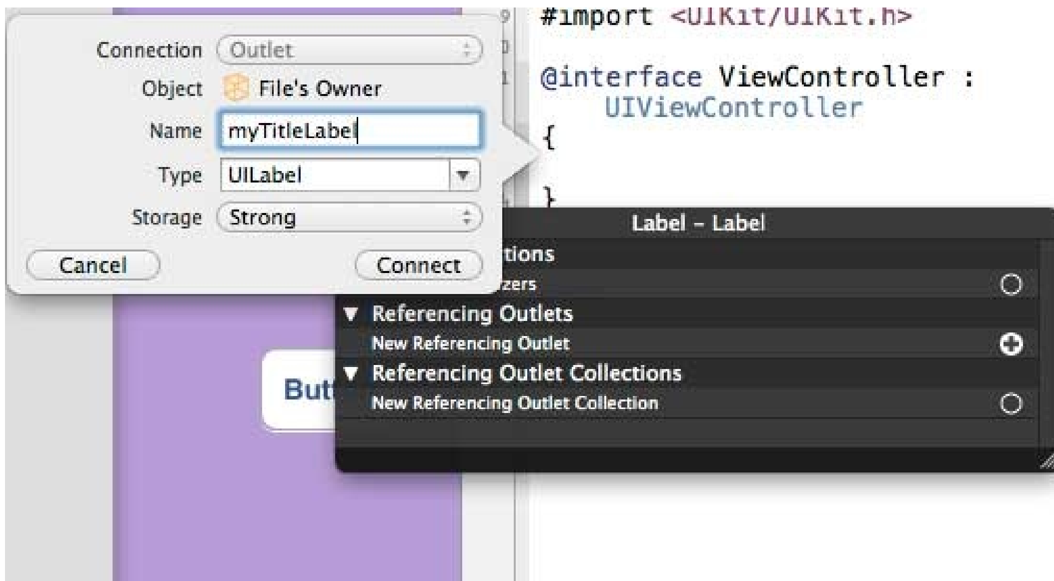


7、编辑器区域中心有两个窗口，ViewController.xib文件和ViewController.h

8、右击标签上的选择按钮，按住并拖动新引用参照，如下所示

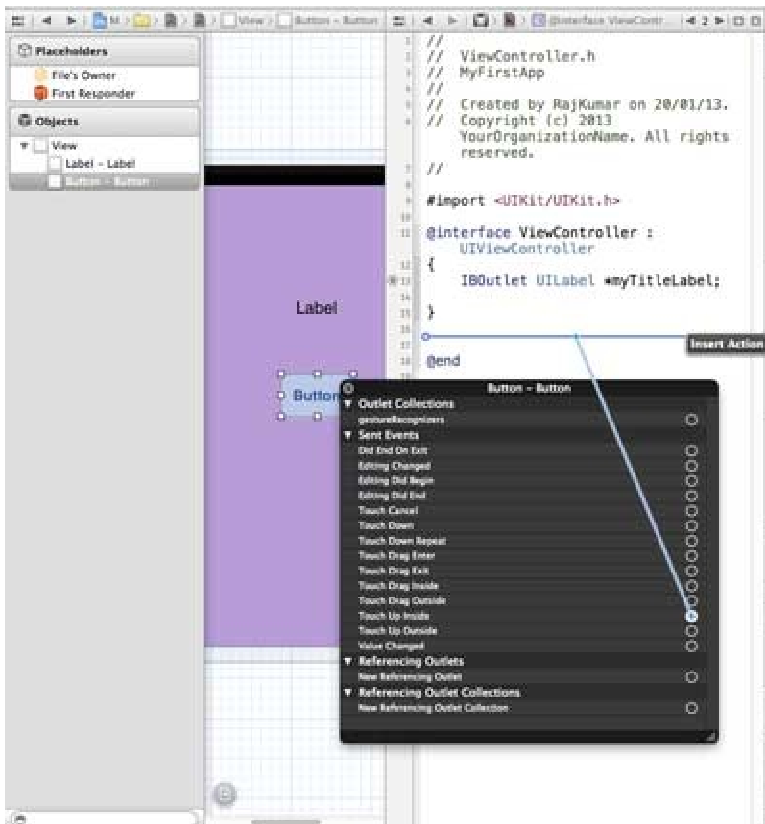


9、现在放在ViewController.h之间的大括号中。也可以放在文件中，如果是这样，必须在做这个之前已经添加了。如下所示

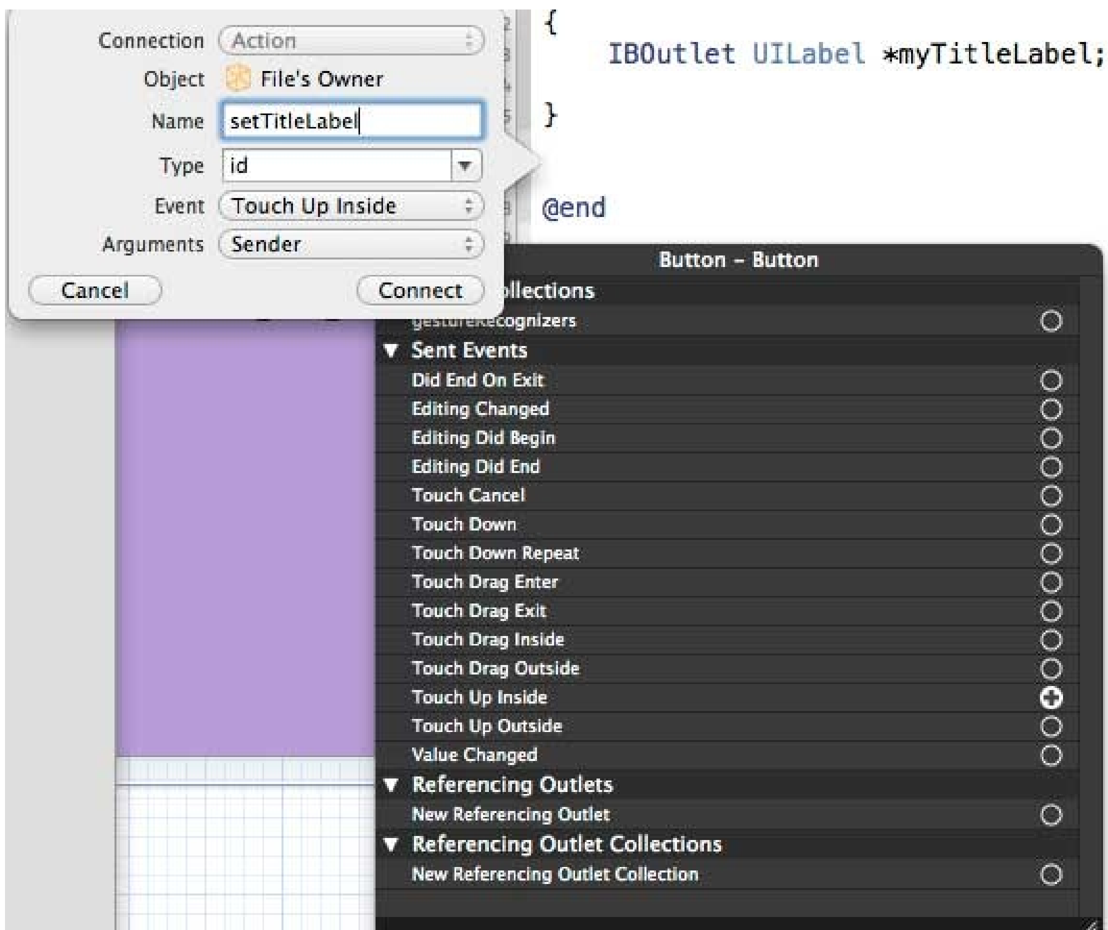


10. 输入输出口（Outlet）的标签名称，这里给出的是myTitleLable。单击链接，完成ibOutlet

11、同样的，添加操作，只需右击倒圆角矩形，选择触摸内心拖动它下方的大括号



12、重新命名为setTitleLable



13、 选择ViewController.m文件，有一种方法，如下所示

```
-(IBAction) setTitleLabel:(id)sender{
}
```

14、在上述的方法内，如下所示，添加一个语句

```
[myTitleLabel setTitleText:@"Hello"];
```

15、选择运行按钮运行该程序，得到如下的输出



16、单击按钮



17.、 创建的参照（outlets）按钮标签已更改为对按钮执行的操作（actions）

18、 由上可知，IBOutlet将创建对UIElement的引用（此处为UILable），同样的IBAction和UIButton通过执行操作和UIButton相链接。

19、 当创建动作时通过选择不同的事件你可以做不同的操作。

委托（Delegates）示例

假设对象A调用B来执行一项操作，操作一旦完成，对象A就必须知道对象B已完成任务且对象A将执行其他必要操作。

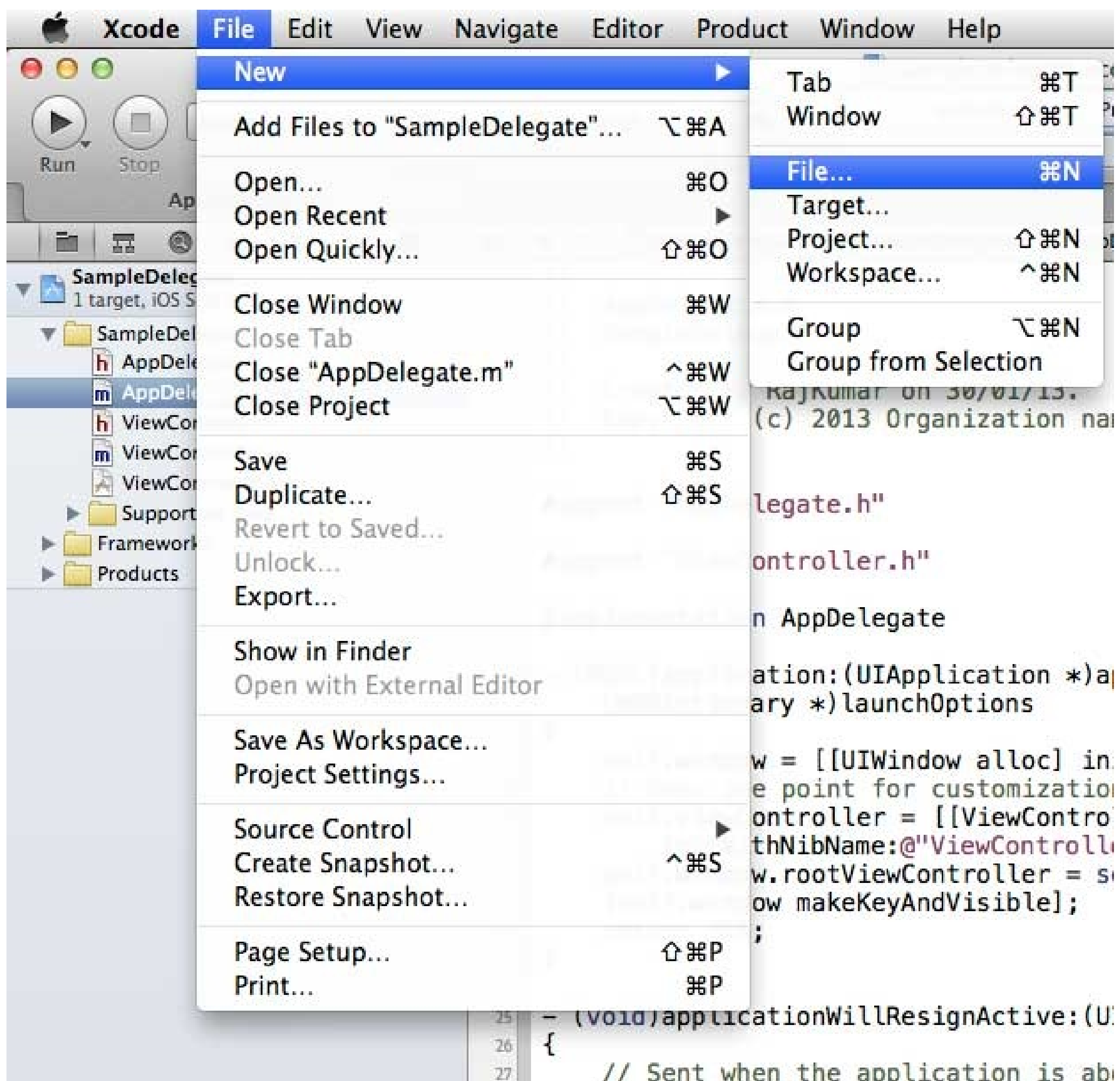
在上面的示例中的关键概念有

- A是B的委托对象
- B引用一个A
- A将实现B的委托方法
- B通过委托方法通知

创建一个委托（Delegates）对象

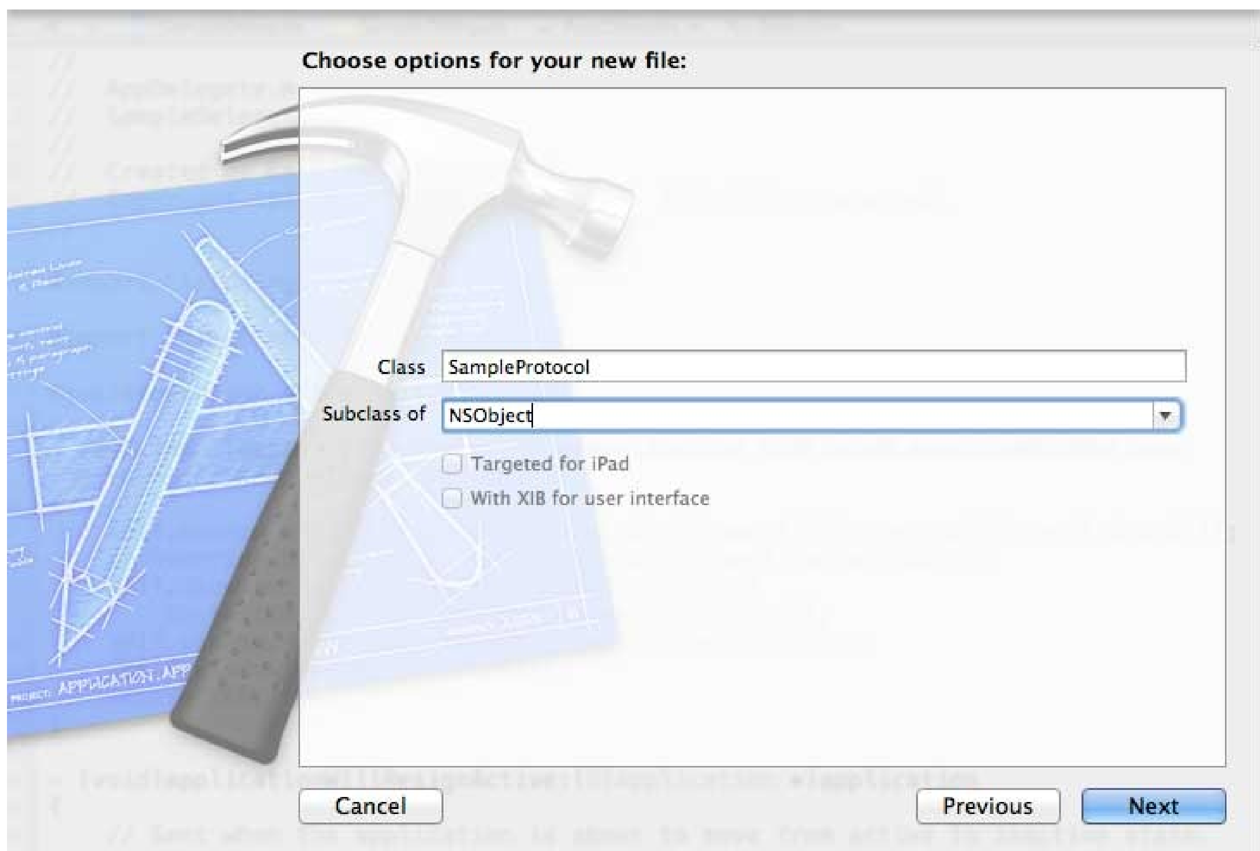
1. 创建一个单一视图的应用程序

2. 然后选择文件 File -> New -> File...



3. 然后选择Objective C单击下一步

4. 将SampleProtocol的子类命名为NSObject，如下所示



5. 然后选择创建

6.向SampleProtocol.h文件夹中添加一种协议，然后更新代码，如下所示：

```
#import &lt;Foundation/Foundation.h&gt;
// Protocol definition starts here
@protocol SampleProtocolDelegate &lt;NSObject&gt;
@required
- (void) processCompleted;
@end
// Protocol Definition ends here
@interface SampleProtocol : NSObject

{
   // Delegate to respond back
   id &lt;SampleProtocolDelegate&gt; _delegate;

}
@property (nonatomic,strong) id delegate;

-(void)startSampleProcess; // Instance method

@end
```

7. Implement the instance method by updating the SampleProtocol.m file as shown below.

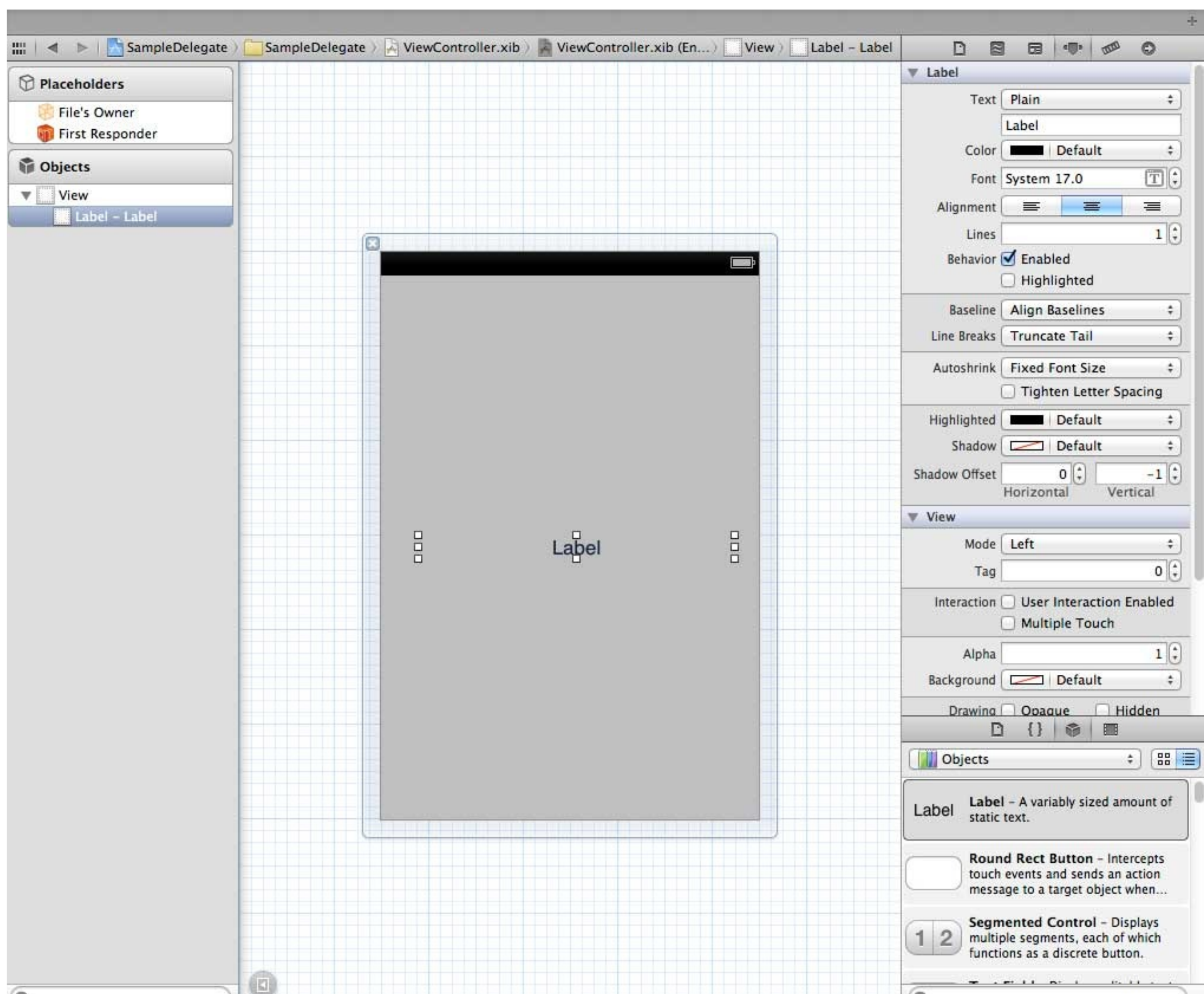
```
#import "SampleProtocol.h"

@implementation SampleProtocol

-(void)startSampleProcess{

    [NSTimer scheduledTimerWithTimeInterval:3.0 target:self.delegat
    selector:@selector(processCompleted) userInfo:nil repeats:NO];
}
@end
```

8. 将标签从对象库拖到UIView，从而在ViewController.xib中添加UILabel，如下所示:



9. 创建一个IBOutlet标签并命名为myLabel，然后按如下所示更新代码并在ViewController.h里显示SampleProtocolDelegate

```
#import &lt;UIKit/UIKit.h&gt;
#import "SampleProtocol.h"

@interface ViewController : UIViewController&lt;SampleProtocolDeleg
{
    IBOutlet UILabel *myLabel;
}
@end
```

10. 完成授权方法，为SampleProtocol创建对象和调用startSampleProcess方法。如下所示，更新ViewController.m文件

```
#import "ViewController.h"

@interface ViewController ()

@end

@implementation ViewController

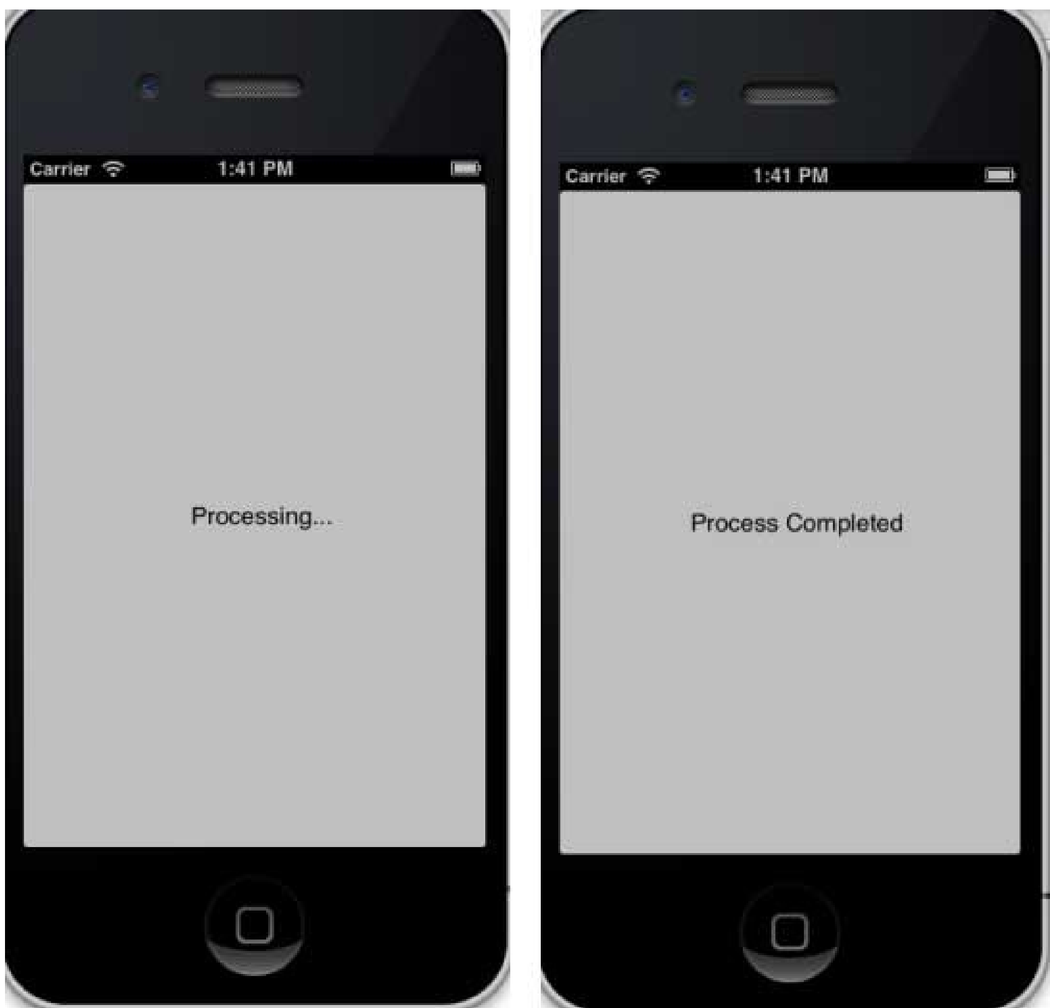
- (void)viewDidLoad
{
    [super viewDidLoad];
    SampleProtocol *sampleProtocol = [[SampleProtocol alloc]init];
    sampleProtocol.delegate = self;
    [myLabel setText:@"Processing..."];
    [sampleProtocol startSampleProcess];
    // Do any additional setup after loading the view, typically fr
}

- (void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}

#pragma mark - Sample protocol delegate
-(void)processCompleted{
    [myLabel setText:@"Process Completed"];
}

@end
```

11. 将看到如下所示的输出结果，最初的标签也会继续运行，一旦授权方法被SampleProtocol对象所调用，标签运行程序的代码也会更新。



什么是UI元素？

UI元素是我们应用程序里可以看见的任何可视元素，其中一些元素响应用户的操作，如按钮、文本字段，有其他的丰富内容，如图像、标签等。

如何添加UI元素？

可以在界面生成器的参与下，在代码中添加UI元素。如果需要，我们可以使用他们其中之一。

我们关注的

通过代码，将集中于添加UI元素到应用程序。比较简单而直接的方法是使用界面生成器拖放UI元素。

方法

以下我们通过创建一款简单的IOS应用程序，来解释一些UI元素

步骤

1、在第一款IOS程序里一样，创建一个Viewbased应用程序

2、只更新ViewController.h和ViewController.m文件

3、然后我们将方法添加到ViewController.m文件中来创建UI元素

4、在viewDidLoad方法中调用此方法

5、重要的代码行已经在代码中通过在单行上方标注的方式进行了注释

用户界面元素列表

下面解释具体的UI元素和其相关的功能

具体的UI元素	功能
Text Fields-文本字段	用户界面元素，使用应用程序来获取用户输入
输入类型-TextFields	用户可以通过使用UITextField来赋予键盘输入属性
Buttons-按钮	用于处理用户操作
Label-标签	用于显示静态内容
Toolbar-工具栏	操纵当前视图所显示的东西
Status Bar-状态栏	显示设备的关键信息
Navigation Bar-导航栏	包含一个可以推断的视图控制器，并弹出导航控制器的导航按钮
Tab bar-选项卡栏	一般用于各个子任务、视图或同一视图中的模型之间的切换.
Image View-图像视图	用于显示一个简单的图像序列
Scroll View-滚动视图	用来显示更多屏幕区域的内容
Table View-列表视图	用于在多个行或部分中显示可滚动列表的数据
IOS分割视图(Split View)	用于在详细信息窗格上显示两个窗格与主窗格的控制信息
Text View-文本视图	用于显示滚动列表的文本信息可以被选中和编辑
View Transition -视图切换	各种视图查看之间的切换
Pickers-选择器	用来显示从列表中选择一个特定的数据
Switches-开关	用作禁用和启用操作
IOS滑块(Sliders)	用来允许用户在允许的值范围内选对一个值
IOS警告对话框(Alerts)	用来给用户提供重要的信息
IOS图标(Icons)	它是图像，表示用于行动或描绘与应用程序相关的东西

文本字段的使用

文本字段是一个用户界面元素，通过应用程序来获取用户输入。

一个UITextfield如下所示：

重要的文本字段的属性

- 在没有任何用户输入时，显示占位符
- 正常文本
- 自动更正型
- 键盘类型
- 返回键类型
- 清除按钮模式
- 对齐方式
- 委托

更新xib中的属性

可以在Utility area（实用区域，窗口的右侧）更改xib在属性查看器中的文本字段属性。



文本字段委托

我们可以通过右击 UIElement 界面生成器中设置委托并将它连接到文件的所有者，如下所示：



使用委托的步骤：

- 1.设置委托如上图所示
- 2.添加委托到您的响应类
- 3.执行文本字段代表，重要的文本字段代表如下:

```
- (void)textFieldDidBeginEditing:(UITextField *)textField
```

```
- (void)textFieldDidEndEditing:(UITextField *)textField
```

- 4.正如其名称所暗示，上述两个委托分别叫做编辑的文本字段和结束编辑
- 5. 其他的委托请查看 UITextDelegate Protocol 参考手册。

实例

以下我们使用简单的实例来创建UI元素

ViewController 类将采用UITextFieldDelegate，修改ViewController.h文件，如下所示:

将方法addTextField添加到我们的 ViewController.m 文件

然后在 viewDidLoad 方法中调用此方法

在ViewController.m中更新viewDidLoad，如下所示

```
#import "ViewController.h"
@interface ViewController ()

@end

@implementation ViewController

- (void)viewDidLoad
{
    [super viewDidLoad];
    //The custom method to create our textfield is called
    [self addTextField];
    // Do any additional setup after loading the view, typically fr
}

- (void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}

-(void)addTextField{
   // This allocates a label
   UILabel *prefixLabel = [[UILabel alloc]initWithFrame:CGRectZero]
   //This sets the label text
   prefixLabel.text =@"## ";
   // This sets the font for the label
   [prefixLabel setFont:[UIFont boldSystemFontOfSize:14]];
   // This fits the frame to size of the text
   [prefixLabel sizeToFit];

   // This allocates the textfield and sets its frame
   UITextField *textField = [[UITextField  alloc] initWithFrame:
   CGRectMake(20, 50, 280, 30)];

   // This sets the border style of the text field
   textField.borderStyle = UITextBorderStyleRoundedRect;
   textField.contentVerticalAlignment =
   UIControlContentVerticalAlignmentCenter;
   [textField setFont:[UIFont boldSystemFontOfSize:12]];

   //Placeholder text is displayed when no text is typed
   textField.placeholder = @"Simple Text field";

   //Prefix label is set as left view and the text starts after tha
   textField.leftView = prefixLabel;
```

```
    //It set when the left prefixLabel to be displayed
    textField.leftViewMode = UITextFieldViewModeAlways;

    // Adds the textField to the view.
    [self.view addSubview:textField];

    // sets the delegate to the current class
    textField.delegate = self;
}

// pragma mark is used for easy access of code in Xcode
#pragma mark - TextField Delegates

// This method is called once we click inside the textField
-(void)textFieldDidBeginEditing:(UITextField *)textField{
    NSLog(@"Text field did begin editing");
}

// This method is called once we complete editing
-(void)textFieldDidEndEditing:(UITextField *)textField{
    NSLog(@"Text field ended editing");
}

// This method enables or disables the processing of return key
-(BOOL) textFieldShouldReturn:(UITextField *)textField{
     [textField resignFirstResponder];
     return YES;
}

- (void)viewDidUnload {
    label = nil;
    [super viewDidUnload];
}

@end
```

运行该应用程序会看到下面的输出

委托调用的方法基于用户操作。要知道调用委托时请参阅控制台输出。

为什么使用不同的输入类型？

键盘输入的类型帮助我们从用户获取必需的输入。

它移除不需要的键，并包括所需的部分。用户可以通过使用 UITextField 的键盘属性设置输入的类型。

- 如：文本字段（textField）。 keyboardType = UIKeyboardTypeDefault

键盘输入类型

输入的类型	描述
UIKeyboardTypeASCIICapable	键盘包括所有标准的 ASCII 字符。
UIKeyboardTypeNumbersAndPunctuation	键盘显示数字和标点。
UIKeyboardTypeURL	键盘的 URL 项优化。
UIKeyboardTypeNumberPad	键盘用于 PIN 输入和显示一个数字键盘。
UIKeyboardTypePhonePad	键盘对输入电话号码进行了优化。
UIKeyboardTypeNamePhonePad	键盘用于输入姓名或电话号码。
UIKeyboardTypeEmailAddress	键盘对输入电子邮件地址的优化。
UIKeyboardTypeDecimalPad	键盘用来输入十进制数字。
UIKeyboardTypeTwitter	键盘对 twitter @ 和 # 符号进行了优化。

添加自定义方法 addTextFieldWithDifferentKeyboard

```
-(void) addTextFieldWithDifferentKeyboard{

    UITextField *textField1= [[UITextField alloc]initWithFrame:
    CGRectMake(20, 50, 280, 30)];
    textField1.delegate = self;
    textField1.borderStyle = UITextBorderStyleRoundedRect;
    textField1.placeholder = @"Default Keyboard";
    [self.view addSubview:textField1];

    UITextField *textField2 = [[UITextField alloc]initWithFrame:
    CGRectMake(20, 100, 280, 30)];
    textField2.delegate = self;
    textField2.borderStyle = UITextBorderStyleRoundedRect;
    textField2.keyboardType = UIKeyboardTypeASCIICapable;
    textField2.placeholder = @"ASCII keyboard";
    [self.view addSubview:textField2];

    UITextField *textField3 = [[UITextField alloc]initWithFrame:
    CGRectMake(20, 150, 280, 30)];
    textField3.delegate = self;
    textField3.borderStyle = UITextBorderStyleRoundedRect;
    textField3.keyboardType = UIKeyboardTypePhonePad;
    textField3.placeholder = @"Phone pad keyboard";
    [self.view addSubview:textField3];

    UITextField *textField4 = [[UITextField alloc]initWithFrame:
    CGRectMake(20, 200, 280, 30)];
    textField4.delegate = self;
    textField4.borderStyle = UITextBorderStyleRoundedRect;
    textField4.keyboardType = UIKeyboardTypeDecimalPad;
    textField4.placeholder = @"Decimal pad keyboard";
    [self.view addSubview:textField4];

    UITextField *textField5= [[UITextField alloc]initWithFrame:
    CGRectMake(20, 250, 280, 30)];
    textField5.delegate = self;
    textField5.borderStyle = UITextBorderStyleRoundedRect;
    textField5.keyboardType = UIKeyboardTypeEmailAddress;
    textField5.placeholder = @"Email keyboard";
    [self.view addSubview:textField5];

    UITextField *textField6= [[UITextField alloc]initWithFrame:
    CGRectMake(20, 300, 280, 30)];
    textField6.delegate = self;
    textField6.borderStyle = UITextBorderStyleRoundedRect;
    textField6.keyboardType = UIKeyboardTypeURL;
    textField6.placeholder = @"URL keyboard";
    [self.view addSubview:textField6];
}
```

在 ViewController.m 中更新 viewDidLoad，如下所示

```
(void)viewDidLoad
{
   [super viewDidLoad];
   //The custom method to create textfield with different keyboard
   [self addTextFieldWithDifferentKeyboard];
   //Do any additional setup after loading the view, typically fron
}
```

输出

现在当我们运行应用程序时我们就会得到下面的输出:

选择不同的文本区域我们将看到不同的键盘。

按钮使用

按钮用于处理用户操作。它截取触摸事件，并将消息发送到目标对象。

圆角矩形按钮

在 xib 中的按钮属性

您可以在Utility area（实用区域，窗口的右侧）的属性检查器的更改 xib 按钮属性。

按钮类型

- UIButtonTypeCustom
- UIButtonTypeRoundedRect
- UIButtonTypeDetailDisclosure
- UIButtonTypeInfoLight
- UIButtonTypeInfoDark
- UIButtonTypeContactAdd

重要的属性

- imageView
- titleLabel

重要的方法

```
+ (id)buttonWithType:(UIButtonType)buttonType
```

```
- (UIImage *)backgroundImageForState:(UIControlState)state
```

```
- (UIImage *)imageForState:(UIControlState)state
```

```
- (void)setTitle:(NSString *)title forState:(UIControlState)state
```

```
- (void)addTarget:(id)target action:(SEL)action forControlEvents: (
```

添加自定义方法 addDifferentTypesOfButton

```
-(void)addDifferentTypesOfButton
{
    // A rounded Rect button created by using class method
    UIButton *roundRectButton = [UIButton buttonWithType:
    UIButtonTypeRoundedRect];
    [roundRectButton setFrame:CGRectMake(60, 50, 200, 40)];
    // sets title for the button
    [roundRectButton setTitle:@"Rounded Rect Button" forState:
    UIControlStateNormal];
    [self.view addSubview:roundRectButton];

    UIButton *customButton = [UIButton buttonWithType: UIButtonType
    [customButton setBackgroundColor: [UIColor lightGrayColor]];
    [customButton setTitleColor:[UIColor blackColor] forState:
    UIControlStateHighlighted];
    //sets background image for normal state
    [customButton setBackgroundImage:[UIImage imageNamed:
    @"Button_Default.png"]
    forState:UIControlStateNormal];
    //sets background image for highlighted state
    [customButton setBackgroundImage:[UIImage imageNamed:
    @"Button_Highlighted.png"]
    forState:UIControlStateHighlighted];
    [customButton setFrame:CGRectMake(60, 100, 200, 40)];
    [customButton setTitle:@"Custom Button" forState:UIControlState
    [self.view addSubview:customButton];

    UIButton *detailDisclosureButton = [UIButton buttonWithType:
    UIButtonTypeDetailDisclosure];
    [detailDisclosureButton setFrame:CGRectMake(60, 150, 200, 40)];
    [detailDisclosureButton setTitle:@"Detail disclosure" forState
    UIControlStateNormal];
    [self.view addSubview:detailDisclosureButton];

    UIButton *contactButton = [UIButton buttonWithType:
    UIButtonTypeContactAdd];
    [contactButton setFrame:CGRectMake(60, 200, 200, 40)];
    [self.view addSubview:contactButton];

    UIButton *infoDarkButton = [UIButton buttonWithType:
    UIButtonTypeInfoDark];
    [infoDarkButton setFrame:CGRectMake(60, 250, 200, 40)];
    [self.view addSubview:infoDarkButton];

    UIButton *infoLightButton = [UIButton buttonWithType:
    UIButtonTypeInfoLight];
    [infoLightButton setFrame:CGRectMake(60, 300, 200, 40)];
    [self.view addSubview:infoLightButton];
}
```

注意：

我们将命名为"Button_Default.png"和"Button_Highlighted.png"的个图像添加到我们的项目，可以通过将图像拖到列出了我们的项目文件的导航区域来完成。

在 ViewController.m 中更新 viewDidLoad，如下所示

```
(void)viewDidLoad
{
    [super viewDidLoad];
    //The custom method to create our different types of button is
    [self addDifferentTypesOfButton];
    //Do any additional setup after loading the view, typically fro
}
```

输出

现在当我们运行应用程序时我们就会得到下面的输出：



标签的使用

标签用于显示静态内容，包括单独的一行或多行。

重要的属性

- textAlignment
- textColor
- text
- numberOflines
- lineBreakMode

添加自定义方法 addLabel

```
-(void)addLabel{
    UILabel *aLabel = [[UILabel alloc]initWithFrame:
    CGRectMake(20, 200, 280, 80)];
    aLabel.numberOfLines = 0;
    aLabel.textColor = [UIColor blueColor];
    aLabel.backgroundColor = [UIColor clearColor];
    aLabel.textAlignment = UITextAlignmentCenter;
    aLabel.text = @"This is a sample text\n of multiple lines.
    here number of lines is not limited.";
    [self.view addSubview:aLabel];
}
```

在 ViewController.m 中更新 viewDidLoad，如下所示:

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    //The custom method to create our label is called
    [self addLabel];
    // Do any additional setup after loading the view, typically fr
}
```

输出

运行应用程序，就会得到下面的输出:



工具栏的使用

我们可以使用工具栏修改视图元素。

如，邮件应用程序里的收件箱栏中有删除、分享、答复等等。如下所示：



重要的属性

- barStyle
- items

添加自定义方法 addToolbar

```
-(void)addToolbar
{
    UIBarButtonItem *spaceItem = [[UIBarButtonItem alloc]
    initWithBarButtonSystemItem:UIBarButtonSystemItemFlexibleSpace
    target:nil action:nil];
    UIBarButtonItem *customItem1 = [[UIBarButtonItem alloc]
    initWithTitle:@"Tool1" style:UIBarButtonItemStyleBordered
    target:self action:@selector(toolBarItem1:)];
    UIBarButtonItem *customItem2 = [[UIBarButtonItem alloc]
    initWithTitle:@"Tool2" style:UIBarButtonItemStyleDone
    target:self action:@selector(toolBarItem2:)];
    NSArray *toolbarItems = [NSArray arrayWithObjects:
    customItem1,spaceItem, customItem2, nil];
    UIToolbar *toolbar = [[UIToolbar alloc]initWithFrame:
    CGRectMake(0, 366+54, 320, 50)];
    [toolbar setBarStyle:UIBarStyleBlackOpaque];
    [self.view addSubview:toolbar];
    [toolbar setItems:toolbarItems];
}
```

为了解所执行的操作我们在我们的ViewController.xib中添加UILabel Iboutlet并为UILabel 创建命名为标签的IBoutlet。

我们还需要添加两个方法来执行，如下所示的工具栏项的操作:

```
-(IBAction)toolBarItem1:(id)sender{
    [label setText:@"Tool 1 Selected"];
}

-(IBAction)toolBarItem2:(id)sender{
    [label setText:@"Tool 2 Selected"];
}
```

在ViewController.m中更新 viewDidLoad，如下所示:

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    // The method hideStatusbar called after 2 seconds
    [self addToolbar];
    // Do any additional setup after loading the view, typically fr
}
```

输出

现在当我们运行该应用程序我们会看到下面的输出。

单击我们得到的 tool1 和 tool2 栏按钮

状态栏的使用

状态栏显示设备的关键信息。

- 设备模型或网络提供商
- 网络信号强度
- 电池使用量
- 时间

状态栏如下所示：

隐藏状态栏的方法

```
[[UIApplication sharedApplication] setStatusBarHidden:YES];
```

另一种隐藏状态栏的方法

我们还可以通过添加行，并在info.plist 的帮助下选择 UIStatusBarHidden 隐藏状态栏，并使其值为否（NO）。

在类中添加自定义方法 hideStatusbar

它隐藏状态栏进行动画处理，并也调整我们认为占据状态栏空间的大小。

```
-(void)hideStatusbar{
    [[UIApplication sharedApplication] setStatusBarHidden:YES
    withAnimation:UIStatusBarAnimationFade];
    [UIView beginAnimations:@"Statusbar hide" context:nil];
    [UIView setAnimationDuration:0.5];
    [self.view setFrame:CGRectMake(0, 0, 320, 480)];
    [UIView commitAnimations];
}
```

在 ViewController.m 中更新 viewDidLoad，如下所示：

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    // The method hideStatusbar called after 2 seconds
    [self performSelector:@selector(hideStatusbar)
    withObject:nil afterDelay:2.0];
    // Do any additional setup after loading the view, typically fi
}
```

初始输出以及2秒后输出

IOS导航栏的使用

导航栏包含导航控制器的导航的按钮。在导航栏中的标题是当前视图控制器的标题。

示例代码和步骤

1.创视图应用程序

1. 现在，选择应用程序 Delegate.h ，添加导航控制器的属性，如下所示:

```
#import <UIKit/UIKit.h>

@class ViewController;

@interface AppDelegate : UIResponder <UIApplicationDelegate>

@property (strong, nonatomic) UIWindow *window;

@property (strong, nonatomic) ViewController *viewController;

@property (strong, nonatomic) UINavigationController *navControlle

@end
```

3. 更新应用程序: didFinishLaunchingWithOptions:方法，在AppDelegate.m文件分配的导航控制器，并使其成为窗口的根视图控制器，如下所示：

```
- (BOOL)application:(UIApplication *)application didFinishLaunching
{
    self.window = [[UIWindow alloc] initWithFrame:
    [[UIScreen mainScreen] bounds]];
    // Override point for customization after application launch.
    self.viewController = [[ViewController alloc]
    initWithNibName:@"ViewController" bundle:nil];
    //Navigation controller init with ViewController as root
    UINavigationController *navController = [[UINavigationControlle
    initWithRootViewController:self.viewController];
    self.window.rootViewController = navController;
    [self.window makeKeyAndVisible];
    return YES;
}
```

4.现在，通过选择**File -> New ->File... ->** Objective C Class 添加新的类文件 TempViewController，然后将类命名 TempViewController 与 UIViewController 的子类。

5.在ViewController.h中添加navButon，如下所示

```
// ViewController.h
#import <UIKit/UIKit.h>

@interface ViewController : UIViewController
{
    UIButton *navButton;
}
@end
```

6.现在添加方法addNavigationBarItem并在viewDidLoad调用方法

7. 为导航项创建方法

1. 我们还需要创建另一种方法到另一视图控制器 TempViewController。
2. 更新后的ViewController.m，如下所示:

```
// ViewController.m
#import "ViewController.h"
#import "TempViewController.h"
@interface ViewController ()

@end
@implementation ViewController

- (void)viewDidLoad
{
    [super viewDidLoad];
    [self addNavigationBarButton];
    //Do any additional setup after loading the view, typically fro
}

- (void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}

-(IBAction)pushNewView:(id)sender{
   TempViewController *tempVC =[[TempViewController alloc]
   initWithNibName:@"TempViewController" bundle:nil];
   [self.navigationController pushViewController:tempVC animated:YE
}

-(IBAction)myButtonClicked:(id)sender{
   // toggle hidden state for navButton
   [navButton setHidden:!nav.hidden];
}

-(void)addNavigationBarButton{
   UIBarButtonItem *myNavBtn = [[UIBarButtonItem alloc] initWithTit
   @"MyButton" style:UIBarButtonItemStyleBordered target:
   self action:@selector(myButtonClicked:)];

   [self.navigationController.navigationBar setBarStyle:UIBarStyleE
   [self.navigationItem setRightBarButtonItem:myNavBtn];

   // create a navigation push button that is initially hidden
   navButton = [UIButton buttonWithType:UIButtonTypeRoundedRect];
   [navButton setFrame:CGRectMake(60, 50, 200, 40)];
   [navButton setTitle:@"Push Navigation" forState:UIControlStateNo
   [navButton addTarget:self action:@selector(pushNewView:)
   forControlEvents:UIControlEventTouchUpInside];
   [self.view addSubview:navButton];
   [navButton setHidden:YES];
}
@end
```

1. 现在当我们运行应用程序时我们就会得到下面的输出



1. 单击 MyButton 导航按钮，切换导航按钮可见性
2. 单击导航按钮，显示另一个视图控制器，如下所示



IOS选项卡栏的使用

它一般用于在同一视图中各个子任务、 视图或的模型之间切换。

选项卡栏的示例如下所示：



重要的属性

- backgroundImage
- items
- selectedItem

示例代码和步骤

1. 创建一个新的项目，选择 **Tabbed Application** 替代视图应用程序 ,点击下一步，输入项目名称和选择 **create**.

1. 这里默认创建两个视图控制器和标签栏添加到我们的应用程序。

3. **AppDelegate.m didFinishLaunchingWithOptions**方法如下：

```
- (BOOL)application:(UIApplication *)application didFinishLaunching
{
    self.window = [[UIWindow alloc] initWithFrame:[[UIScreen mainSc
    bounds]];
    // Override point for customization after application launch.
    UIViewController *viewController1 = [[FirstViewController alloc
    initWithNibName:@"FirstViewController" bundle:nil];
    UIViewController *viewController2 = [[SecondViewController allo
    initWithNibName:@"SecondViewController" bundle:nil];
    self.tabBarController = [[UITabBarController alloc] init];
    self.tabBarController.viewControllers = @[viewController1,
    viewController2];
    self.window.rootViewController = self.tabBarController;
    [self.window makeKeyAndVisible];
    return YES;
}
```

1. 两个视图控制器被用来分配作为选项卡栏控制器的视图控制器
2. 运行应用程序,得到如下结果：



图像视图的使用

图像视图用于显示单个图像或动画序列的图像。

重要的属性

- image
- highlightedImage
- userInteractionEnabled
- animationImages
- animationRepeatCount

重要的方法

```
- (id)initWithImage:(UIImage *)image
```

```
- (id)initWithImage:(UIImage *)image highlightedImage:
  (UIImage *)highlightedImage
```

```
- (void)startAnimating
```

```
- (void)stopAnimating
```

添加自定义方法 addImageView

```
-(void)addImageView{
    UIImageView *imgview = [[UIImageView alloc]
    initWithFrame:CGRectMake(10, 10, 300, 400)];
    [imgview setImage:[UIImage imageNamed:@"AppleUSA1.jpg"]];
    [imgview setContentMode:UIViewContentModeScaleAspectFit];
    [self.view addSubview:imgview];
}
```

添加另一个自定义方法 addImageViewWithAnimation

这种方法解释了如何对imageView 中的图像进行动画处理

```
-(void)addImageViewWithAnimation{
    UIImageView *imgview = [[UIImageView alloc]
    initWithFrame:CGRectMake(10, 10, 300, 400)];
    // set an animation
    imgview.animationImages = [NSArray arrayWithObjects:
    [UIImage imageNamed:@"AppleUSA1.jpg"],
    [UIImage imageNamed:@"AppleUSA2.jpg"], nil];
    imgview.animationDuration = 4.0;
    imgview.contentMode = UIViewContentModeCenter;
    [imgview startAnimating];
    [self.view addSubview:imgview];
}
```

注意： 我们必须添加命名为"AppleUSA1.jpg"和"AppleUSA2.jpg"到我们的项目，可以通过将图像拖到我们导航区域，其中列出了我们的项目文件所做的图像。

在 ViewController.m 中更新 viewDidLoad，如下所示

```
(void)viewDidLoad
{
   [super viewDidLoad];
   [self addImageView];
}
```

滚动视图的使用

如果内容超出屏幕的大小就会使用到滚动视图来显示隐藏的部分。

它可以包含所有的其他用户界面元素 如图像视图、 标签、 文本视图甚至另一个滚动视图。

重要的属性

- contentSize
- contentInset
- contentOffset
- delegate

重要的方法

```
- (void)scrollRectToVisible:(CGRect)rect animated:(BOOL)animated
```

```
- (void)setContentOffset:(CGPoint)contentOffset animated:(BOOL)anir
```

重要的委托方法

在ViewController.h中，加入<uiscrollviewdelegate>滚动视图和声明滚动视图让类符合委托协议，如下所示:</uiscrollviewdelegate>

```
#import <UIKit/UIKit.h>

@interface ViewController : UIViewController<UIScrollViewDelegate>
{
    UIScrollView *myScrollView;
}

@end
```

添加自定义方法 addScrollView

```
-(void)addScrollView{
    myScrollView = [[UIScrollView alloc]initWithFrame:
    CGRectMake(20, 20, 280, 420)];
    myScrollView.accessibilityActivationPoint = CGPointMake(100, 10
    imgView = [[UIImageView alloc]initWithImage:
    [UIImage imageNamed:@"AppleUSA.jpg"]];
    [myScrollView addSubview:imgView];
    myScrollView.minimumZoomScale = 0.5;
    myScrollView.maximumZoomScale = 3;
    myScrollView.contentSize = CGSizeMake(imgView.frame.size.width,
    imgView.frame.size.height);
    myScrollView.delegate = self;
    [self.view addSubview:myScrollView];
}
```

注意： 我们必须添加一个命名为"AppleUSA1.jpg"到我们的项目，可以通过将图像拖到我们导航区域，其中列出了我们的项目文件所做的图像。图像应高于设备的高度。

ViewController.m中实现scrollView 委托

```
 -(UIView *)viewForZoomingInScrollView:(UIScrollView *)scrollView{
    return imgView;
}
-(void)scrollViewDidEndDecelerating:(UIScrollView *)scrollView{
    NSLog(@"Did end decelerating");
}
-(void)scrollViewDidScroll:(UIScrollView *)scrollView{
//    NSLog(@"Did scroll");
}
-(void)scrollViewDidEndDragging:(UIScrollView *)scrollView
  willDecelerate:(BOOL)decelerate{
    NSLog(@"Did end dragging");
}
-(void)scrollViewWillBeginDecelerating:(UIScrollView *)scrollView{
    NSLog(@"Did begin decelerating");
}
-(void)scrollViewWillBeginDragging:(UIScrollView *)scrollView{
    NSLog(@"Did begin dragging");
}
```

在 **ViewController.m** 中更新 **viewDidLoad**，如下所示

```
(void)viewDidLoad
{
   [super viewDidLoad];
   [self addScrollView];
   //Do any additional setup after loading the view, typically fror
}
```

输出

现在当我们运行该应用程序我们会看到下面的输出。一旦滚动滚动视图，将能够查看图像的其余部分:



表格视图的使用

IOS表格视图由单元格 （一般可重复使用） 组成，用于显示垂直滚动的视图。

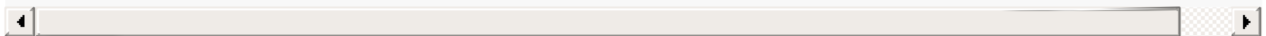
在iOS 中,表格视图用于显示数据列表,如联系人、待办事项或购物项列表。

重要的属性

- delegate
- dataSource
- rowHeight
- sectionFooterHeight
- sectionHeaderHeight
- separatorColor
- tableHeaderView
- tableFooterView

重要的方法

```
- (UITableViewCell *)cellForRowAtIndexPath:(NSIndexPath *)indexPath
```



```
- (void)deleteRowsAtIndexPaths:(NSArray *)indexPaths
  withRowAnimation:(UITableViewRowAnimation)animation
```

```
- (id)dequeueReusableCellWithIdentifier:(NSString *)identifier
```

```
- (id)dequeueReusableCellWithIdentifier:(NSString *)identifier
  forIndexPath:(NSIndexPath *)indexPath
```

```
- (void)reloadData
```

```
- (void)reloadRowsAtIndexPaths:(NSArray *)indexPaths
  withRowAnimation:(UITableViewRowAnimation)animation
```

```
- (NSArray *)visibleCells
```

示例代码和步骤

1.在ViewController.xib中添加表格视图，如下所示



2. 通过右键单击并选择数据源和委托将委托和数据源设定到"File's Owner（文件的所有者）"。设置数据源如下所示



3.为表格视图创建IBOutlet的并将其命名为myTableView。如以下图片中所示





4. 为拥有数据，添加一个NSMutableArray使其能够在列表格视图中显示

5.ViewController应采用的UITableViewDataSource和UITableViewDelegate协议。ViewController.h代码如下所示

```
#import <UIKit/UIKit.h>

@interface ViewController : UIViewController<UITableViewDataSource,
UITableViewDelegate>
{

    IBOutlet UITableView *myTableView;
    NSMutableArray *myData;
}

@end
```

6.执行所需的表格视图委托和数据源的方法。更新ViewController.m，如下所示

```
#import "ViewController.h"

@interface ViewController ()

@end

@implementation ViewController

- (void)viewDidLoad
{
    [super viewDidLoad];
    // table view data is being set here
    myData = [[NSMutableArray alloc]initWithObjects:
    @"Data 1 in array",@"Data 2 in array",@"Data 3 in array",
    @"Data 4 in array",@"Data 5 in array",@"Data 5 in array",
    @"Data 6 in array",@"Data 7 in array",@"Data 8 in array",
    @"Data 9 in array", nil];
    // Do any additional setup after loading the view, typically fr
}

- (void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}
```

```
#pragma mark - Table View Data source
- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSecti
  (NSInteger)section{
    return [myData count]/2;
}

- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowA
  (NSIndexPath *)indexPath{
    static NSString *cellIdentifier = @"cellID";

    UITableViewCell *cell = [tableView dequeueReusableCellWithIdent
    cellIdentifier];
    if (cell == nil) {
        cell = [[UITableViewCell alloc]initWithStyle:
        UITableViewCellStyleDefault reuseIdentifier:cellIdentifier]
    }
    NSString *stringForCell;
    if (indexPath.section == 0) {
        stringForCell= [myData objectAtIndex:indexPath.row];

    }
    else if (indexPath.section == 1){
        stringForCell= [myData objectAtIndex:indexPath.row+ [myData

    }
    [cell.textLabel setText:stringForCell];
    return cell;
}

// Default is 1 if not implemented
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView{
    return 2;
}

- (NSString *)tableView:(UITableView *)tableView titleForHeaderInSe
  (NSInteger)section{
    NSString *headerTitle;
    if (section==0) {
        headerTitle = @"Section 1 Header";
    }
    else{
        headerTitle = @"Section 2 Header";

    }
    return headerTitle;
}
- (NSString *)tableView:(UITableView *)tableView titleForFooterInSe
  (NSInteger)section{
    NSString *footerTitle;
    if (section==0) {
        footerTitle = @"Section 1 Footer";
    }
```

```
    else{
        footerTitle = @"Section 2 Footer";

    }
    return footerTitle;
}

#pragma mark - TableView delegate

-(void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:
 (NSIndexPath *)indexPath{
    [tableView deselectRowAtIndexPath:indexPath animated:YES];
    UITableViewCell *cell = [tableView cellForRowAtIndexPath:indexP
    NSLog(@"Section:%d Row:%d selected and its data is %@",
    indexPath.section,indexPath.row,cell.textLabel.text);
}

@end
```

7.现在当我们运行应用程序时我们就会得到下面的输出



分割视图的使用

分割视图是 iPad 的特定视图控制器用于管理两个视图控制器，在左侧是一个主控制器，其右侧是一个详细信息视图控制器。 重要的属性

- delegate
- viewControllers

示例代码和步骤

1.创建一个新项目，选择Master Detail Application并单击下一步，输入项目名称，然后选择创建。

2.简单的分割视图控制器与母版中的表视图是默认创建的。

3.在这里我们为我们创建的下列文件。

- AppDelegate.h
- AppDelegate.m
- DetailViewController.h
- DetailViewController.m
- DetailViewController.xib
- MasterViewController.h
- MasterViewController.m
- MasterViewController.xib

4. AppDelegate.h文件如下所示

```
#import <UIKit/UIKit.h>

@interface AppDelegate : UIResponder <UIApplicationDelegate>

@property (strong, nonatomic) UIWindow *window;

@property (strong, nonatomic) UISplitViewController *splitViewContr

@end
```

5.在AppDelegate.m中的didFinishLaunchingWithOptions方法，如下所示

```
  - (BOOL)application:(UIApplication *)application
    didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
    self.window = [[UIWindow alloc] initWithFrame:[[UIScreen mainScr
    bounds]];
     // Override point for customization after application launch.
     MasterViewController *masterViewController = [[MasterViewContro
     alloc] initWithNibName:@"MasterViewController" bundle:nil];
     UINavigationController *masterNavigationController =
     [[UINavigationController alloc] initWithRootViewController:
     masterViewController];

     DetailViewController *detailViewController =
     [[DetailViewController alloc] initWithNibName:@"DetailViewContr
     bundle:nil];
     UINavigationController *detailNavigationController =
     [[UINavigationController alloc] initWithRootViewController:
     detailViewController];

     masterViewController.detailViewController = detailViewControlle

     self.splitViewController = [[UISplitViewController alloc] init]
     self.splitViewController.delegate = detailViewController;
     self.splitViewController.viewControllers =
     @[masterNavigationController, detailNavigationController];
     self.window.rootViewController = self.splitViewController;
     [self.window makeKeyAndVisible];
     return YES;
}
```

6. MasterViewController.h，如下所示

```
#import <UIKit/UIKit.h>

@class DetailViewController;

@interface MasterViewController : UITableViewController

@property (strong, nonatomic) DetailViewController *detailViewCont

@end
```

7. MasterViewController.m，如下所示

```
#import "MasterViewController.h"

#import "DetailViewController.h"

@interface MasterViewController () {
    NSMutableArray *_objects;
}
@end

@implementation MasterViewController

- (id)initWithNibName:(NSString *)nibNameOrNil bundle:(NSBundle *)
  nibBundleOrNil
{
    self = [super initWithNibName:nibNameOrNil bundle:nibBundleOrN
    if (self) {
        self.title = NSLocalizedString(@"Master", @"Master");
        self.clearsSelectionOnViewWillAppear = NO;
        self.contentSizeForViewInPopover = CGSizeMake(320.0, 600.0
    }
    return self;
}

- (void)viewDidLoad
{
    [super viewDidLoad];
    self.navigationItem.leftBarButtonItem = self.editButtonItem;

    UIBarButtonItem *addButton = [[UIBarButtonItem alloc]
    initWithBarButtonSystemItem: UIBarButtonSystemItemAdd
    target:self action:@selector(insertNewObject:)];
    self.navigationItem.rightBarButtonItem = addButton;
}

- (void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
```

```
}

- (void)insertNewObject:(id)sender
{
    if (!_objects) {
        _objects = [[NSMutableArray alloc] init];
    }
    [_objects insertObject:[NSDate date] atIndex:0];
    NSIndexPath *indexPath = [NSIndexPath indexPathForRow:0 inSecti
    [self.tableView insertRowsAtIndexPaths:@[indexPath] withRowAnim
    UITableViewRowAnimationAutomatic];
}

#pragma mark - Table View

- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView
{
    return 1;
}

- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSecti
  (NSInteger)section
{
    return _objects.count;
}

// Customize the appearance of table view cells.
- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowA
  (NSIndexPath *)indexPath
{
    static NSString *CellIdentifier = @"Cell";

    UITableViewCell *cell = [tableView dequeueReusableCellWithIdent
    CellIdentifier];
    if (cell == nil) {
        cell = [[UITableViewCell alloc] initWithStyle:
        UITableViewCellStyleDefault reuseIdentifier:CellIdentifier]
    }

    NSDate *object = _objects[indexPath.row];
    cell.textLabel.text = [object description];
    return cell;
}

- (BOOL)tableView:(UITableView *)tableView canEditRowAtIndexPath:
  (NSIndexPath *)indexPath
{
    // Return NO if you do not want the specified item to be editab
    return YES;
}

- (void)tableView:(UITableView *)tableView commitEditingStyle:
  (UITableViewCellEditingStyle)editingStyle forRowAtIndexPath:
```

```
  (NSIndexPath *)indexPath
{
    if (editingStyle == UITableViewCellEditingStyleDelete) {
        [_objects removeObjectAtIndex:indexPath.row];
        [tableView deleteRowsAtIndexPaths:@[indexPath] withRowAnima
        UITableViewRowAnimationFade];
    } else if (editingStyle == UITableViewCellEditingStyleInsert) {
        // Create a new instance of the appropriate class, insert i
        //the array, and add a new row to the table view.
    }
}

/*
// Override to support rearranging the table view.
- (void)tableView:(UITableView *)tableView moveRowAtIndexPath:
  (NSIndexPath *) fromIndexPath toIndexPath:(NSIndexPath *)toIndexP
{
}
*/

/*
// Override to support conditional rearranging of the table view.
- (BOOL)tableView:(UITableView *)tableView canMoveRowAtIndexPath:
  (NSIndexPath *)indexPath
{
    // Return NO if you do not want the item to be re-orderable.
    return YES;
}
*/

- (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:
  (NSIndexPath *)indexPath
{
    NSDate *object = _objects[indexPath.row];
    self.detailViewController.detailItem = object;
    NSDateFormatter *formatter = [[NSDateFormatter alloc] init];
    [formatter setDateFormat: @"yyyy-MM-dd HH:mm:ss zzz"];
    NSString *stringFromDate = [formatter stringFromDate:object];
    self.detailViewController.detailDescriptionLabel.text = stringF
}

@end
```

8. DetailViewController.h， 如下所示

```
#import <UIKit/UIKit.h>

@interface DetailViewController : UIViewController
<UISplitViewControllerDelegate>

@property (strong, nonatomic) id detailItem;

@property (weak, nonatomic) IBOutlet UILabel *detailDescriptionLabe
@end
```

9. DetailViewController.m， 如下所示

```
#import "DetailViewController.h"

@interface DetailViewController ()
@property (strong, nonatomic) UIPopoverController *masterPopoverCo
- (void)configureView;
@end

@implementation DetailViewController

#pragma mark - Managing the detail item

- (void)setDetailItem:(id)newDetailItem
{
    if (_detailItem != newDetailItem) {
        _detailItem = newDetailItem;

        // Update the view.
        [self configureView];
    }

    if (self.masterPopoverController != nil) {
        [self.masterPopoverController dismissPopoverAnimated:YES];
    }
}

- (void)configureView
{
    // Update the user interface for the detail item.

    if (self.detailItem) {
        self.detailDescriptionLabel.text = [self.detailItem descri
    }
}

- (void)viewDidLoad
{
    [super viewDidLoad];
    [self configureView];
```

```
}

- (void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}

- (id)initWithNibName:(NSString *)nibNameOrNil bundle:
  (NSBundle *)nibBundleOrNil
{
    self = [super initWithNibName:nibNameOrNil bundle:nibBundleOrNi
    if (self) {
        self.title = NSLocalizedString(@"Detail", @"Detail");
    }
    return self;
}

#pragma mark - Split view

- (void)splitViewController:(UISplitViewController *)splitControlle
  willHideViewController:(UIViewController *)viewController withBar
  (UIBarButtonItem *)barButtonItem forPopoverController:
  (UIPopoverController *)popoverController
{
    barButtonItem.title = NSLocalizedString(@"Master", @"Master");
    [self.navigationItem setLeftBarButtonItem:barButtonItem animate
    self.masterPopoverController = popoverController;
}

- (void)splitViewController:(UISplitViewController *)splitControlle
  willShowViewController:(UIViewController *)viewController
  invalidatingBarButtonItem:(UIBarButtonItem *)barButtonItem
{
    // Called when the view is shown again in the split view,
    //invalidating the button and popover controller.
    [self.navigationItem setLeftBarButtonItem:nil animated:YES];
    self.masterPopoverController = nil;
}

@end
```

10.现在当我们运行应用程序时，在横向模式下我们会得到下面的输出

11. 当我们切换到纵向模式，我们会获得下面的输出:

IOS文本视图的使用

文本视图用于显示多行滚动的文本。

重要属性

- dataDetectorTypes
- delegate
- editable
- inputAccessoryView
- inputView
- text
- textAlignment
- textColor

重要的委托方法

```
-(void)textViewDidBeginEditing:(UITextView *)textView
```

```
-(void)textViewDidEndEditing:(UITextView *)textView
```

```
-(void)textViewDidChange:(UITextView *)textView
```

```
-(BOOL)textViewShouldEndEditing:(UITextView *)textView
```

添加自定义方法 **addTextView**

```
-(void)addTextView{
    myTextView = [[UITextView alloc]initWithFrame:
    CGRectMake(10, 50, 300, 200)];
    [myTextView setText:@"Lorem ipsum dolor sit er elit lamet, cons
    cillium adipisicing pecu, sed do eiusmod tempor incididunt ut l
    dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exer
    ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis a
    dolor in reprehenderit in voluptate velit esse cillum dolore eu
    nulla pariatur. Excepteur sint occaecat cupidatat non proident,
    culpa qui officia deserunt mollit anim id est laborum. Nam libe
    conscient to factor tum poen legum odioque civiuda.
    Lorem ipsum dolor sit er elit lamet, consectetaur cillium adipi
    pecu, sed do eiusmod tempor incididunt ut labore et dolore magn
    Ut enim ad minim veniam, quis nostrud exercitation ullamco labo
    aliquip ex ea commodo consequat. Duis aute irure dolor in repre
    in voluptate velit esse cillum dolore eu fugiat nulla pariatur.
    Excepteur sint occaecat cupidatat non proident, sunt in culpa
    qui officia deserunt mollit anim id est laborum. Nam liber te c
    to factor tum poen legum odioque civiuda."];
    myTextView.delegate = self;
    [self.view addSubview:myTextView];

}
```

在 ViewController.m 中执行 textView 委托

```
#pragma mark - Text View delegates

-(BOOL)textView:(UITextView *)textView shouldChangeTextInRange:
 (NSRange)range replacementText:(NSString *)text{
    if ([text isEqualToString:@"\n"]) {
        [textView resignFirstResponder];
    }
    return YES;
}
-(void)textViewDidBeginEditing:(UITextView *)textView{
    NSLog(@"Did begin editing");
}
-(void)textViewDidChange:(UITextView *)textView{
    NSLog(@"Did Change");

}
-(void)textViewDidEndEditing:(UITextView *)textView{
    NSLog(@"Did End editing");

}
-(BOOL)textViewShouldEndEditing:(UITextView *)textView{
    [textView resignFirstResponder];
    return YES;
}
```

修改 ViewController.m 中的 viewDidLoad方法，如下所示

```
(void)viewDidLoad
{
   [super viewDidLoad];
   [self addTextView];
}
```

结果输出

现在当我们运行该应用程序我们会看到下面的输出

IOS视图切换的使用

视图切换通过一系列动画效果实现,包括折叠切换、爆炸切换、卡片式切换等等。

修改 ViewController.xib，如下所示

在 xib 中创建按钮的操作

修改 ViewController.h

```
#import <UIKit/UIKit.h>

@interface ViewController : UIViewController
{
    UIView *view1;
    UIView *view2;
}

-(IBAction)flipFromLeft:(id)sender;
-(IBAction)flipFromRight:(id)sender;
-(IBAction)flipFromTop:(id)sender;
-(IBAction)flipFromBottom:(id)sender;
-(IBAction)curlUp:(id)sender;
-(IBAction)curlDown:(id)sender;
-(IBAction)dissolve:(id)sender;
-(IBAction)noTransition:(id)sender;

@end
```

在 ViewController 类中声明两个视图的实例。ViewController.h文件代码如下：

修改 ViewController.m

我们将添加自定义方法setUpView来初始化视图。

我们还将创建了另一种方法doTransitionWithType: 实现view1切换到view2，反之亦然。

后我们将执行之前创建的操作的方法即调用 doTransitionWithType: 方法与切换类型。ViewController.m代码如下:

```
#import "ViewController.h"

@interface ViewController ()

@end

@implementation ViewController

- (void)viewDidLoad
{
    [super viewDidLoad];
    [self setUpView];
    // Do any additional setup after loading the view, typically fr
}
```

```
-(void)setUpView{
    view1 = [[UIView alloc]initWithFrame:self.view.frame];
    view1.backgroundColor = [UIColor lightTextColor];
    view2 = [[UIView alloc]initWithFrame:self.view.frame];
    view2.backgroundColor = [UIColor orangeColor];
    [self.view addSubview:view1];
    [self.view sendSubviewToBack:view1];

}

-(void)doTransitionWithType:(UIViewAnimationTransition)animationTra
    if ([[self.view subviews] containsObject:view2 ]) {
        [UIView transitionFromView:view2
                            toView:view1
                          duration:2
                           options:animationTransitionType
                        completion:^(BOOL finished){
                            [view2 removeFromSuperview];
                        }];
        [self.view addSubview:view1];
        [self.view sendSubviewToBack:view1];
    }
    else{

        [UIView transitionFromView:view1
                            toView:view2
                          duration:2
                           options:animationTransitionType
                        completion:^(BOOL finished){
                            [view1 removeFromSuperview];
                        }];
        [self.view addSubview:view2];
        [self.view sendSubviewToBack:view2];

    }
}

-(IBAction)flipFromLeft:(id)sender
{
    [self doTransitionWithType:UIViewAnimationOptionTransitionFlipF

}
-(IBAction)flipFromRight:(id)sender{
    [self doTransitionWithType:UIViewAnimationOptionTransitionFlipF

}
-(IBAction)flipFromTop:(id)sender{
    [self doTransitionWithType:UIViewAnimationOptionTransitionFlipF

}
-(IBAction)flipFromBottom:(id)sender{
    [self doTransitionWithType:UIViewAnimationOptionTransitionFlipF
```

```
}
-(IBAction)curlUp:(id)sender{
    [self doTransitionWithType:UIViewAnimationOptionTransitionCurlU

}
-(IBAction)curlDown:(id)sender{
    [self doTransitionWithType:UIViewAnimationOptionTransitionCurlD

}
-(IBAction)dissolve:(id)sender{
    [self doTransitionWithType:UIViewAnimationOptionTransitionCross

}
-(IBAction)noTransition:(id)sender{
    [self doTransitionWithType:UIViewAnimationOptionTransitionNone]

}

- (void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}

@end
```

输出

现在当我们运行该应用程序我们会看到下面的输出

您可以选择不同的按钮，看切换是如何工作。选择蜷缩切换将效果如下所示

选择器的使用

选择器是一个可滚动视图，用于选取列表项中的值。

重要的属性

- delegate
- dataSource

重要的方法

```
- (void)reloadAllComponents
```

```
- (void)reloadComponent:(NSInteger)component
```

```
- (NSInteger)selectedRowInComponent:(NSInteger)component
```

```
- (void)selectRow:(NSInteger)row inComponent:(NSInteger)component
  animated:(BOOL)animated
```

修改 ViewController.h

我们将添加一个文本字段、选择器视图和一个数组。

我们将采用UITextFieldDelegate、UIPickerViewDataSource、UIPickerViewDelegate的协议。ViewController.h文件代码如下所示：

```
#import <UIKit/UIKit.h>

@interface ViewController : UIViewController
<UITextFieldDelegate,UIPickerViewDataSource,UIPickerViewDelegate>
{

    UITextField *myTextField;
    UIPickerView *myPickerView;
    NSArray *pickerArray;
}
@end
```

添加自定义方法 addPickerView

```
-(void)addPickerView{
    pickerArray = [[NSArray alloc]initWithObjects:@"Chess",
    @"Cricket",@"Football",@"Tennis",@"Volleyball", nil];
    myTextField = [[UITextField alloc]initWithFrame:
    CGRectMake(10, 100, 300, 30)];
    myTextField.borderStyle = UITextBorderStyleRoundedRect;
    myTextField.textAlignment = UITextAlignmentCenter;
    myTextField.delegate = self;
    [self.view addSubview:myTextField];
    [myTextField setPlaceholder:@"Pick a Sport"];
    myPickerView = [[UIPickerView alloc]init];
    myPickerView.dataSource = self;
    myPickerView.delegate = self;
    myPickerView.showsSelectionIndicator = YES;
    UIBarButtonItem *doneButton = [[UIBarButtonItem alloc]
    initWithTitle:@"Done" style:UIBarButtonItemStyleDone
    target:self action:@selector(done:)];
    UIToolbar *toolBar = [[UIToolbar alloc]initWithFrame:
    CGRectMake(0, self.view.frame.size.height-
    myDatePicker.frame.size.height-50, 320, 50)];
    [toolBar setBarStyle:UIBarStyleBlackOpaque];
    NSArray *toolbarItems = [NSArray arrayWithObjects:
    doneButton, nil];
    [toolBar setItems:toolbarItems];
    myTextField.inputView = myPickerView;
    myTextField.inputAccessoryView = toolBar;

}
```

执行委托，如下所示:

```
#pragma mark - Text field delegates

-(void)textFieldDidBeginEditing:(UITextField *)textField{
    if ([textField.text isEqualToString:@""]) {
        [self dateChanged:nil];
    }
}
#pragma mark - Picker View Data source
-(NSInteger)numberOfComponentsInPickerView:(UIPickerView *)pickerVi
    return 1;
}
-(NSInteger)pickerView:(UIPickerView *)pickerView
 numberOfRowsInComponent:(NSInteger)component{
    return [pickerArray count];
}

#pragma mark- Picker View Delegate

-(void)pickerView:(UIPickerView *)pickerView didSelectRow:
 (NSInteger)row inComponent:(NSInteger)component{
    [myTextField setText:[pickerArray objectAtIndex:row]];
}
- (NSString *)pickerView:(UIPickerView *)pickerView titleForRow:
  (NSInteger)row forComponent:(NSInteger)component{
    return [pickerArray objectAtIndex:row];
}
```

在ViewController.m修改viewDidLoad，如下所示:

```
(void)viewDidLoad
{
   [super viewDidLoad];
   [self addPickerView];
}
```

输出

现在当我们运行该应用程序我们会看到下面的输出:

文本选择器视图如下所示，我们可以选取我们需要的值：

IOS开关的使用

开关用于打开和关闭状态之间的切换。

重要的属性

- onImage
- offImage
- on

重要的方法

```
- (void)setOn:(BOOL)on animated:(BOOL)animated
```

添加自定义方法 **addSwitch** 和开关

```
-(IBAction)switched:(id)sender{
    NSLog(@"Switch current state %@", mySwitch.on ? @"On" : @"Off")
}
-(void)addSwitch{
    mySwitch = [[UISwitch alloc] init];
    [self.view addSubview:mySwitch];
    mySwitch.center = CGPointMake(150, 200);
    [mySwitch addTarget:self action:@selector(switched:)
    forControlEvents:UIControlEventValueChanged];
}
```

在 **ViewController.m** 中修改 **viewDidLoad**，如下所示

```
(void)viewDidLoad
{
   [super viewDidLoad];
   [self addSwitch];
}
```

输出

现在当我们运行该应用程序我们会看到下面的输出

向右滑动开关输出如下所示

IOS滑块的使用

滑块用于从某个范围的值里选择的一个值。

重要的属性

- continuous
- maximumValue
- minimumValue
- value

重要的方法

```
- (void)setValue:(float)value animated:(BOOL)animated
```

添加自定义方法 **addSlider** 和 **sliderChanged**

```
-(IBAction)sliderChanged:(id)sender{
    NSLog(@"SliderValue %f",mySlider.value);
}
-(void)addSlider{
    mySlider = [[UISlider alloc] initWithFrame:CGRectMake(50, 200,
    [self.view addSubview:mySlider];
    mySlider.minimumValue = 10.0;
    mySlider.maximumValue = 99.0;
    mySlider.continuous = NO;
    [mySlider addTarget:self action:@selector(sliderChanged:)
    forControlEvents:UIControlEventValueChanged];
}
```

在 **ViewController.m** 中修改 **viewDidLoad**，如下所示

```
(void)viewDidLoad
{
   [super viewDidLoad];
   [self addSlider];
}
```

输出

现在当我们运行该应用程序我们会看到下面的输出

当拖动滑块效果如下：



IOS警告对话框的使用

警告对话框用来给用户提供重要信息。

仅在警告对话框视图中选择选项后，才能着手进一步使用应用程序。

重要的属性

- alertViewStyle
- cancelButtonIndex
- delegate
- message
- numberOfButtons
- title

重要的方法

```
- (NSInteger)addButtonWithTitle:(NSString *)title
```

```
- (NSString *)buttonTitleAtIndex:(NSInteger)buttonIndex
```

```
- (void)dismissWithClickedButtonIndex:
  (NSInteger)buttonIndex animated:(BOOL)animated
```

```
- (id)initWithTitle:(NSString *)title message:
  (NSString *)message delegate:(id)delegate
  cancelButtonTitle:(NSString *)cancelButtonTitle
  otherButtonTitles:(NSString*)otherButtonTitles, ...
```

```
- (void)show
```

更新 **ViewController.h**，如下所示

让类符合警告对话框视图的委托协议，如下所示，在ViewController.h中添加 <UIAlertViewDelegate>

```
#import <UIKit/UIKit.h>

@interface ViewController : UIViewController<UIAlertViewDelegate>{

}
@end
```

添加自定义方法 addAlertView

```
-(void)addAlertView{
    UIAlertView *alertView = [[UIAlertView alloc]initWithTitle:
    @"Title" message:@"This is a test alert" delegate:self
    cancelButtonTitle:@"Cancel" otherButtonTitles:@"Ok", nil];
    [alertView show];
}
```

执行警告对话框视图的委托方法

```
#pragma mark - Alert view delegate
- (void)alertView:(UIAlertView *)alertView clickedButtonAtIndex:
  (NSInteger)buttonIndex{
    switch (buttonIndex) {
        case 0:
            NSLog(@"Cancel button clicked");
            break;
        case 1:
            NSLog(@"OK button clicked");
            break;

        default:
            break;
    }
}
```

在 ViewController.m 中修改 viewDidLoad，如下所示

```
(void)viewDidLoad
{
   [super viewDidLoad];
   [self addAlertView];
}
```

输出

现在当我们运行该应用程序我们会看到下面的输出：



IOS图标的使用

IOS图标是用于应用程序相关的操作。

IOS 中的不同图标

- AppIcon
- App Store 的应用程序图标
- 搜索结果和设置的小图标
- 工具栏和导航栏图标
- 选项卡栏图标

AppIcon

AppIcon 是出现在设备SpringBoard （默认屏幕上的所有的应用程序） 的应用程序的图标。

App Store 的应用程序图标

它是512 x 512 或 1024 x 1024(推荐大小)的高分辨率的应用程序图标。

搜索结果和设置的小图标

在搜索列表的应用程序中使用这个小图标。

它还用于与相关的应用程序的功能是启用和禁用的设置屏幕上。如：启用定位服务。

工具栏和导航栏图标

工具栏和导航栏中使用特制的标准图标的列表。它包括的份额，像图标相机，撰写等等。

选项卡栏图标

选项卡栏中使用一系列特制的标准图标列表。它包括的图标有书签、 联系人、 下载等。

有的不同的 iOS 设备的每个图标大小的都不一样。你可以查看更多关于苹果文件中图标的准则：ios人机交互界面指南。

IOS加速度传感器(accelerometer)

简介

加速度传感器是根据x、y和z三个方向来检测在设备位置的改变。

通过加速度传感器可以知道当前设备相对于地面的位置。

以下实例代码需要在真实设备上运行，在模拟器上是无法工作的。

实例步骤

1、创建一个简单的视图应用程序

2、在ViewController.xib中添加三个标签，并创建一个ibOutlets分别为：xlable、ylabel和zlabel

3、如下所示，更新ViewController.h

```
#import <UIKit/UIKit.h>

@interface ViewController : UIViewController<UIAccelerometerDelegat
{
    IBOutlet UILabel *xlabel;
    IBOutlet UILabel *ylabel;
    IBOutlet UILabel *zlabel;

}
@end
```

4、如下所示，更新ViewController.m

```
#import "ViewController.h"

@interface ViewController ()

@end

@implementation ViewController

- (void)viewDidLoad
{
    [super viewDidLoad];
    [[UIAccelerometer sharedAccelerometer]setDelegate:self];
    //Do any additional setup after loading the view,typically fror
}

- (void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}
- (void)accelerometer:(UIAccelerometer *)accelerometer didAccelerat
  (UIAcceleration *)acceleration{
    [xlabel setText:[NSString stringWithFormat:@"%f",acceleration.x
    [ylabel setText:[NSString stringWithFormat:@"%f",acceleration.y
    [zlabel setText:[NSString stringWithFormat:@"%f",acceleration.z
}

@end
```

输出

当我们在iPhone设备中运行该应用程序，得到的输出结果如下所示。


No SIM
9:01 AM
87%
-0.036697
-0.886536
-0.421844

IOS通用应用程序

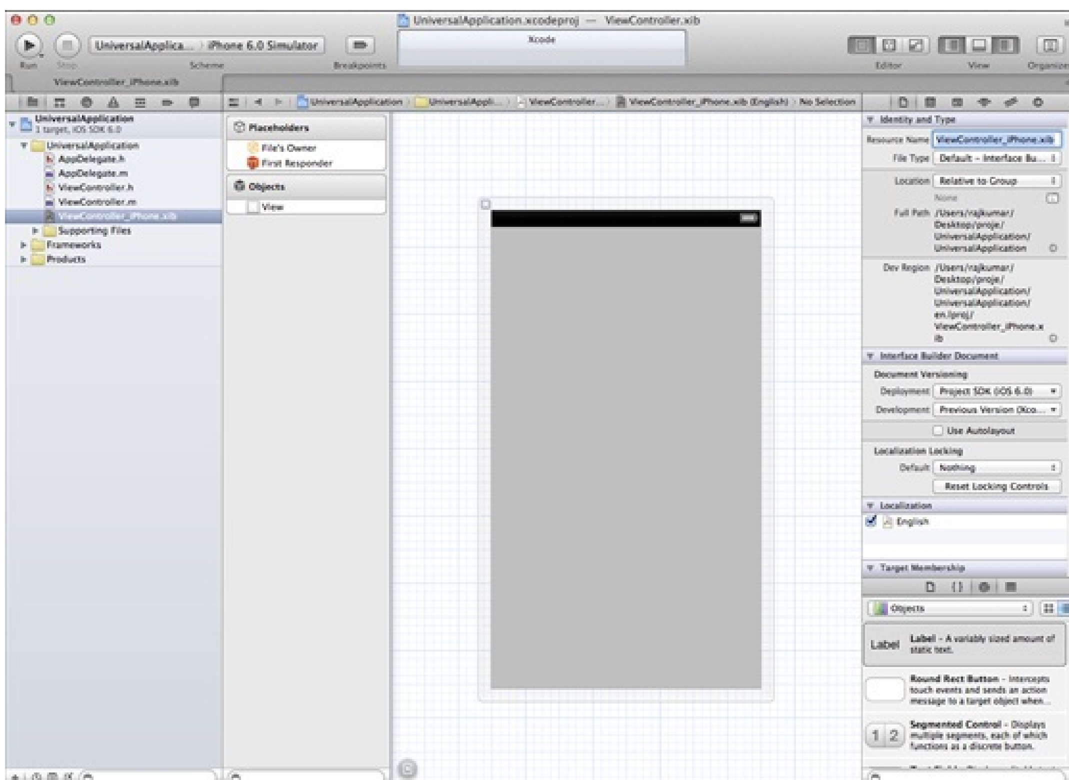
简介

通用的应用程序是为iPhone和iPad在一个单一的二进制文件中设计的应用程序。这有助于代码重用，并能够帮助更快进行更新。

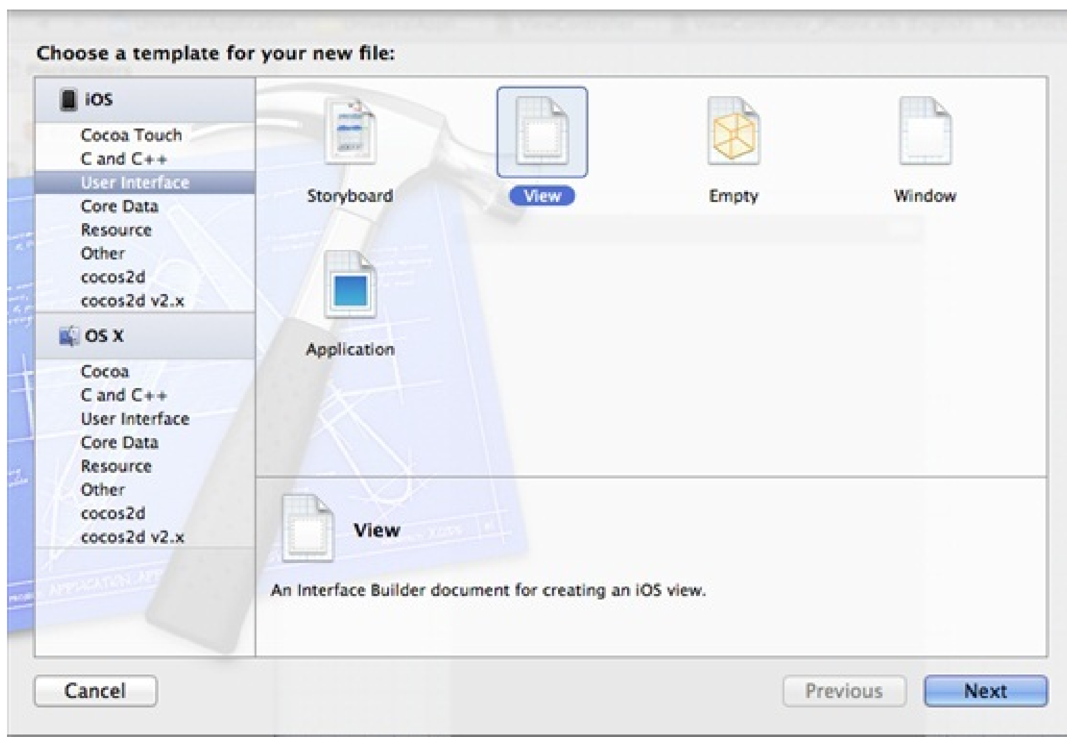
实例步骤

1、创建一个简单的View based application（视图应用程序）

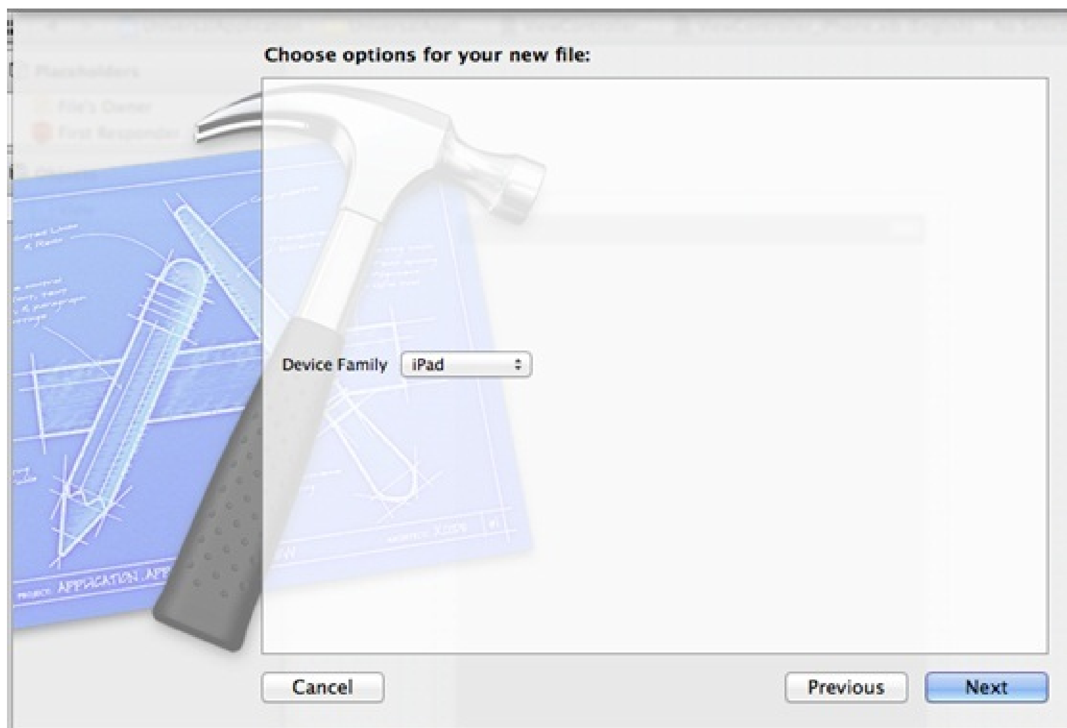
2、在文件查看器的右边，将文件ViewController.xib的文件名称更改为ViewController_iPhone.xib，如下所示



3、选择"File -> New -> File... "，然后选择User Interface，再选择View，单击下一步



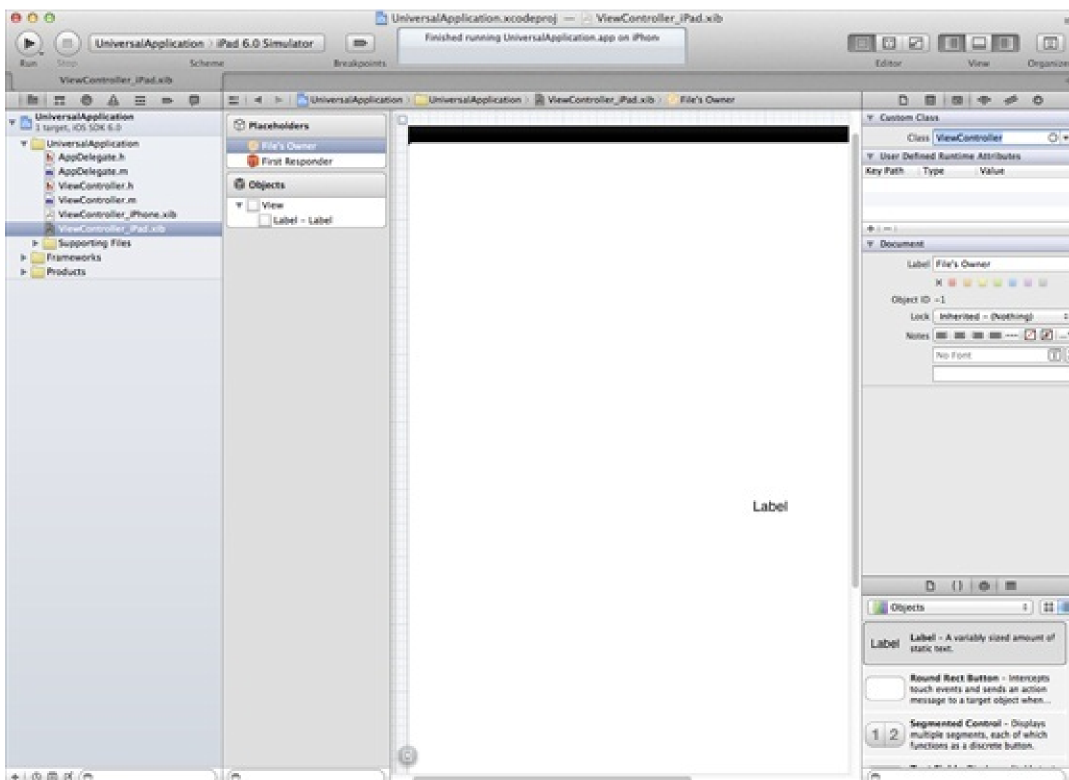
4、选择iPad作为设备，单击下一步:



5、将该文件另存为ViewController_iPad.xib，然后选择创建

6、在ViewController_iPhone.xib和ViewController_iPad.xibd的屏幕中心添加标签

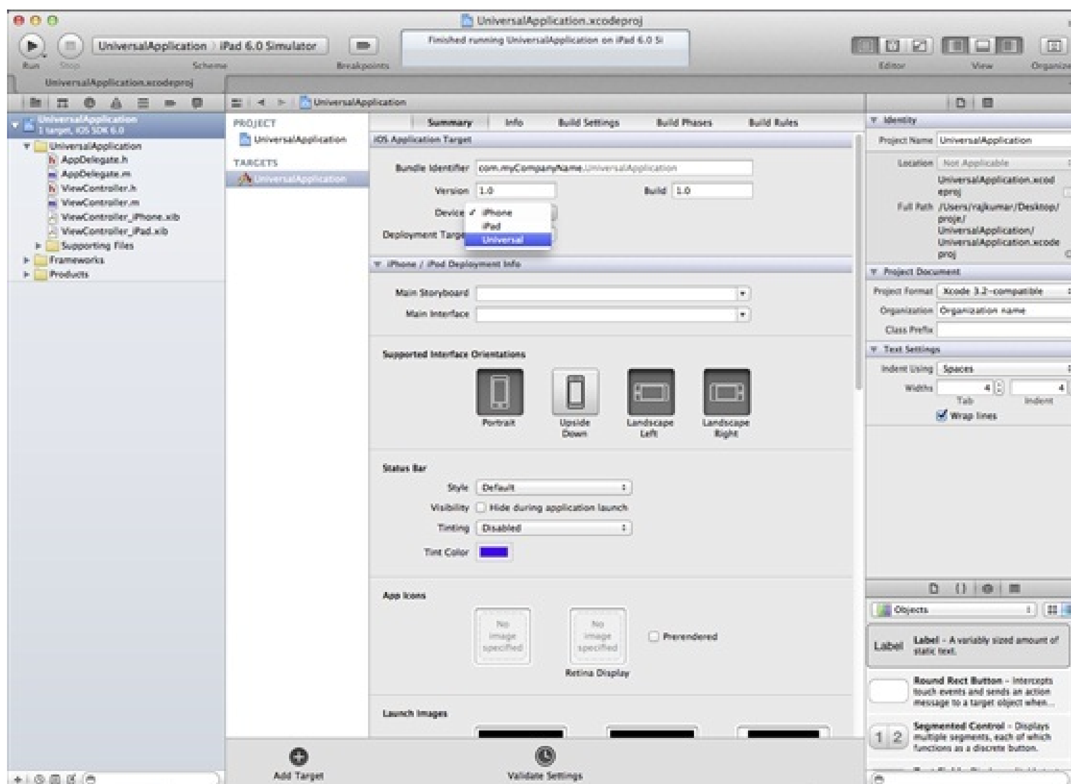
7、在ViewController_iPhone.xib中选择identity inspector，设置custom class为ViewController



8、更新AppDelegate.m中的 application:DidFinishLaunching:withOptions方法

```
- (BOOL)application:(UIApplication *)application
  didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
   self.window = [[UIWindow alloc] initWithFrame:[[UIScreen
   mainScreen] bounds]];
   // Override point for customization after application launch.
   if (UI_USER_INTERFACE_IDIOM() == UIUserInterfaceIdiomPhone) {
        self.viewController = [[ViewController alloc]
        initWithNibName:@"ViewController_iPhone" bundle:nil];
   }
   else{
        self.viewController = [[ViewController alloc] initWithNibNa
        @"ViewController_iPad" bundle:nil];
   }
   self.window.rootViewController = self.viewController;
   [self.window makeKeyAndVisible];
   return YES;
}
```

9、在项目摘要中更新设备中为universal，如下所示：

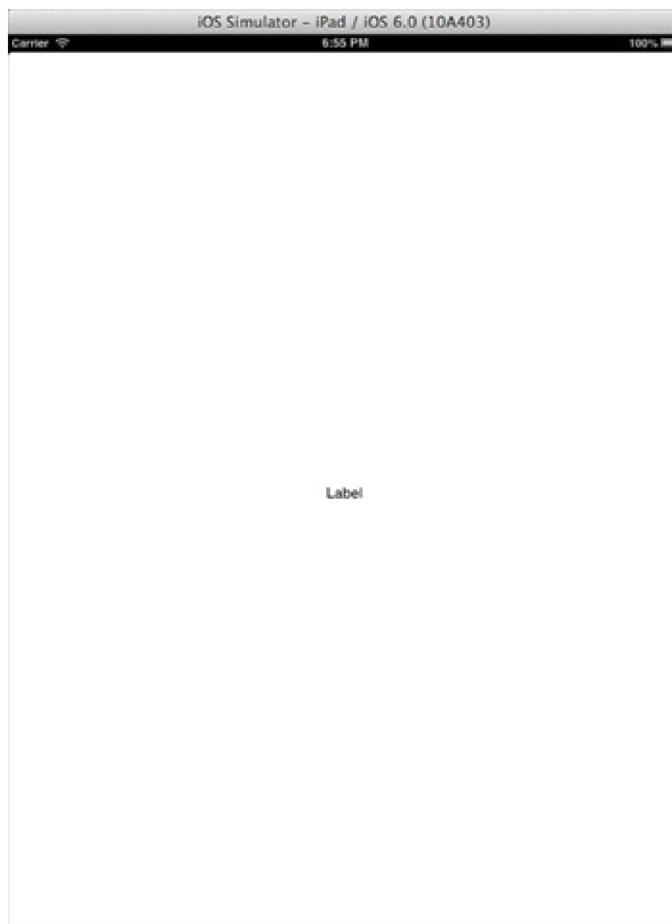


输出

运行该应用程序，我们会看到下面的输出



在iPad模拟器中运行应用程序,我们会得到下面的输出:



IOS相机管理

相机简介

相机是移动设备的共同特点之一，我们能够使用相机拍摄图片，并在应用程序里调用它，而且相机的使用很简单。

实例步骤

1、创建一个简单的View based application

2、在ViewController.xib中添加一个button （按钮），并为该按钮创建IBAction

3、添加一个 image view （图像视图），并创建一个名为imageView的IBOutlet

4、ViewController.h文件代码如下所示：

```
#import <UIKit/UIKit.h>

@interface ViewController : UIViewController<UIImagePickerControlle
{
   UIImagePickerController *imagePicker;
   IBOutlet UIImageView *imageView;
}
- (IBAction)showCamera:(id)sender;

@end
```

5、修改ViewController.m,如下所示：

```
#import "ViewController.h"

@interface ViewController ()

@end

@implementation ViewController

- (void)viewDidLoad
{
    [super viewDidLoad];
}

- (void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}

- (IBAction)showCamera:(id)sender {
    imagePicker.allowsEditing = YES;
    if ([UIImagePickerController isSourceTypeAvailable:
        UIImagePickerControllerSourceTypeCamera])
    {
        imagePicker.sourceType = UIImagePickerControllerSourceType(
    }
    else{
        imagePicker.sourceType =
        UIImagePickerControllerSourceTypePhotoLibrary;
    }
    [self presentModalViewController:imagePicker animated:YES];

}
-(void)imagePickerController:(UIImagePickerController *)picker
  didFinishPickingMediaWithInfo:(NSDictionary *)info{
    UIImage *image = [info objectForKey:UIImagePickerControllerEdit
    if (image == nil) {
        image = [info objectForKey:UIImagePickerControllerOriginalI
    }
    imageView.image = image;

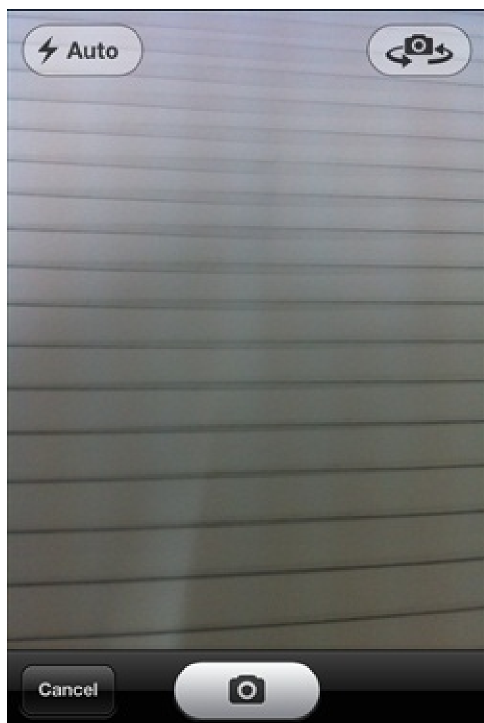
}

-(void)imagePickerControllerDidCancel:(UIImagePickerController *)p
    [self dismissModalViewControllerAnimated:YES];
}

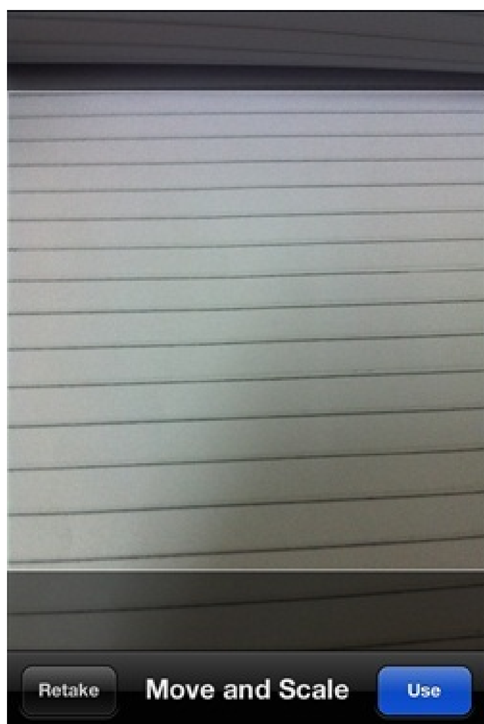
@end
```

输出

运行该应用程序并单击显示相机按钮时，我们就会获得下面的输出



只要拍照之后，就可以通过移动和缩放对图片进行编辑，如下所示。



IOS定位操作

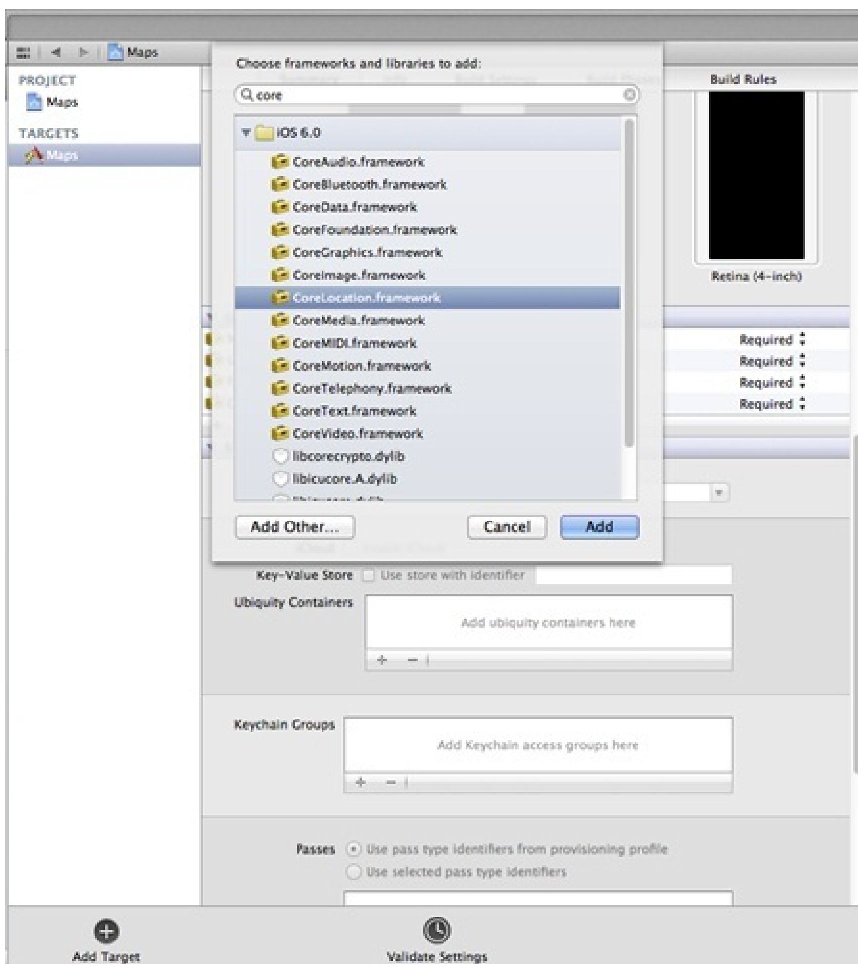
简介

在IOS中通过CoreLocation定位，可以获取到用户当前位置，同时能得到装置移动信息。

实例步骤

1、创建一个简单的View based application（视图应用程序）。

2、择项目文件，然后选择目标，然后添加CoreLocation.framework,如下所示



3、在ViewController.xib中添加两个标签，创建ibOutlet名为latitudeLabel和longtitudeLabel的标签

4、现在通过选择" File-> New -> File... -> "选择Objective C class 并单击下一步

5、把"sub class of"作为NSObject，将类命名为LocationHandler

6、选择创建

7、更新LocationHandler.h，如下所示

```
#import <Foundation/Foundation.h>
#import <CoreLocation/CoreLocation.h>

@protocol LocationHandlerDelegate <NSObject>

@required
-(void) didUpdateToLocation:(CLLocation*)newLocation
 fromLocation:(CLLocation*)oldLocation;
@end

@interface LocationHandler : NSObject<CLLocationManagerDelegate>
{
    CLLocationManager *locationManager;
}
@property(nonatomic,strong) id<LocationHandlerDelegate> delegate;

+(id)getSharedInstance;
-(void)startUpdating;
-(void) stopUpdating;

@end
```

8、更新LocationHandler.m,如下所示

```
#import "LocationHandler.h"
static LocationHandler *DefaultManager = nil;

@interface LocationHandler()

-(void)initiate;

@end

@implementation LocationHandler

+(id)getSharedInstance{
    if (!DefaultManager) {
        DefaultManager = [[self allocWithZone:NULL]init];
        [DefaultManager initiate];
    }
    return DefaultManager;
}
-(void)initiate{
    locationManager = [[CLLocationManager alloc]init];
    locationManager.delegate = self;
}

-(void)startUpdating{
    [locationManager startUpdatingLocation];
}

-(void) stopUpdating{
    [locationManager stopUpdatingLocation];
}
-(void)locationManager:(CLLocationManager *)manager didUpdateToLoca
 (CLLocation *)newLocation fromLocation:(CLLocation *)oldLocation{
    if ([self.delegate respondsToSelector:@selector
    (didUpdateToLocation:fromLocation:)])
    {
        [self.delegate didUpdateToLocation:oldLocation
        fromLocation:newLocation];

    }
}

@end
```

9、更新ViewController.h,如下所示

```
#import <UIKit/UIKit.h>
#import "LocationHandler.h"
@interface ViewController : UIViewController<LocationHandlerDelegat
{
    IBOutlet UILabel *latitudeLabel;
    IBOutlet UILabel *longitudeLabel;
}
@end
```

10、更新ViewController.m,如下所示

```
#import "ViewController.h"

@interface ViewController ()

@end

@implementation ViewController

- (void)viewDidLoad
{
    [super viewDidLoad];
    [[LocationHandler getSharedInstance]setDelegate:self];
    [[LocationHandler getSharedInstance]startUpdating];
}

- (void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}

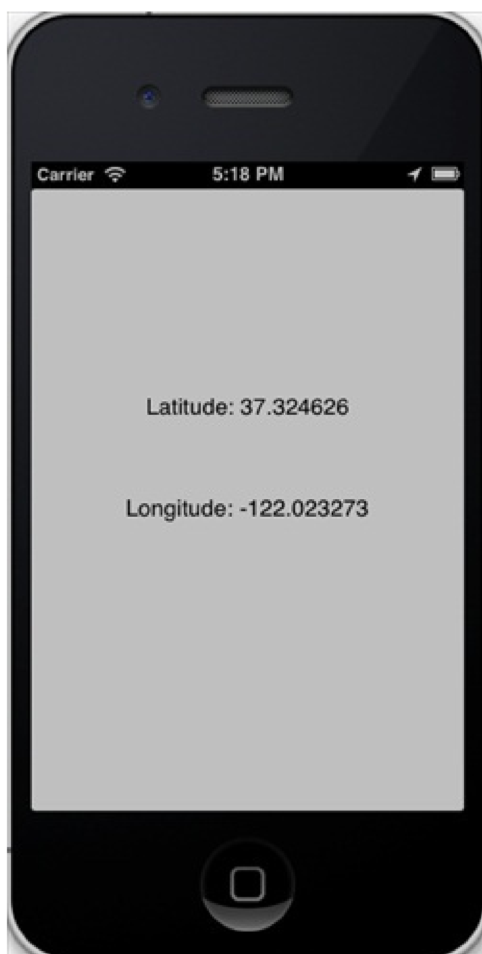
-(void)didUpdateToLocation:(CLLocation *)newLocation
 fromLocation:(CLLocation *)oldLocation{
    [latitudeLabel setText:[NSString stringWithFormat:
    @"Latitude: %f",newLocation.coordinate.latitude]];
    [longitudeLabel setText:[NSString stringWithFormat:
    @"Longitude: %f",newLocation.coordinate.longitude]];

}

@end
```

输出

当我们运行该应用程序，会得到如下的输出:


Carrier
5:18 PM
Latitude: 37.324626
Longitude: -122.023273

IOS SQLite数据库

简介

在IOS中使用Sqlite来处理数据。如果你已经了解了SQL，那你可以很容易的掌握SQLite数据库的操作。

实例步骤

1、创建一个简单的View based application

2、选择项目文件，然后选择目标，添加libsqlite3.dylib库到选择框架

3、通过选择" File-> New -> File... -> "选择 Objective C class 创建新文件，单击下一步

4、"sub class of"为NSObject"，类命名为DBManager

5、选择创建

6、更新DBManager.h,如下所示

```
#import <Foundation/Foundation.h>
#import <sqlite3.h>

@interface DBManager : NSObject
{
    NSString *databasePath;
}

+(DBManager*)getSharedInstance;
-(BOOL)createDB;
-(BOOL) saveData:(NSString*)registerNumber name:(NSString*)name
  department:(NSString*)department year:(NSString*)year;
-(NSArray*) findByRegisterNumber:(NSString*)registerNumber;

@end
```

7、更新DBManager.m,如下所示

```
#import "DBManager.h"
static DBManager *sharedInstance = nil;
static sqlite3 *database = nil;
static sqlite3_stmt *statement = nil;

@implementation DBManager
```

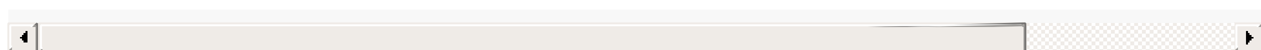
```
+(DBManager*)getSharedInstance{
    if (!sharedInstance) {
        sharedInstance = [[super allocWithZone:NULL]init];
        [sharedInstance createDB];
    }
    return sharedInstance;
}

-(BOOL)createDB{
    NSString *docsDir;
    NSArray *dirPaths;
    // Get the documents directory
    dirPaths = NSSearchPathForDirectoriesInDomains
    (NSDocumentDirectory, NSUserDomainMask, YES);
    docsDir = dirPaths[0];
    // Build the path to the database file
    databasePath = [[NSString alloc] initWithString:
    [docsDir stringByAppendingPathComponent: @"student.db"]];
    BOOL isSuccess = YES;
    NSFileManager *filemgr = [NSFileManager defaultManager];
    if ([filemgr fileExistsAtPath: databasePath ] == NO)
    {
        const char *dbpath = [databasePath UTF8String];
        if (sqlite3_open(dbpath, &database) == SQLITE_OK)
        {
            char *errMsg;
            const char *sql_stmt =
            "create table if not exists studentsDetail (regno inte
            primary key, name text, department text, year text)";
            if (sqlite3_exec(database, sql_stmt, NULL, NULL, &errM
               != SQLITE_OK)
            {
                isSuccess = NO;
                NSLog(@"Failed to create table");
            }
            sqlite3_close(database);
            return  isSuccess;
        }
        else {
            isSuccess = NO;
            NSLog(@"Failed to open/create database");
        }
    }
    return isSuccess;
}

- (BOOL) saveData:(NSString*)registerNumber name:(NSString*)name
  department:(NSString*)department year:(NSString*)year;
{
    const char *dbpath = [databasePath UTF8String];
    if (sqlite3_open(dbpath, &database) == SQLITE_OK)
    {
```

```
        NSString *insertSQL = [NSString stringWithFormat:@"insert i
        studentsDetail (regno,name, department, year) values
        (\"%d\",\"%@\", \"%@\", \"%@\")",[registerNumber integerVal
        name, department, year];
        const char *insert_stmt = [insertSQL UTF8String];
        sqlite3_prepare_v2(database, insert_stmt,-1, &statement, NU
        if (sqlite3_step(statement) == SQLITE_DONE)
        {
            return YES;
        }
        else {
            return NO;
        }
        sqlite3_reset(statement);
    }
    return NO;
}

- (NSArray*) findByRegisterNumber:(NSString*)registerNumber
{
    const char *dbpath = [databasePath UTF8String];
    if (sqlite3_open(dbpath, &database) == SQLITE_OK)
    {
        NSString *querySQL = [NSString stringWithFormat:
        @"select name, department, year from studentsDetail where
        regno=\"%@\"",registerNumber];
        const char *query_stmt = [querySQL UTF8String];
        NSMutableArray *resultArray = [[NSMutableArray alloc]init];
        if (sqlite3_prepare_v2(database,
           query_stmt, -1, &statement, NULL) == SQLITE_OK)
        {
            if (sqlite3_step(statement) == SQLITE_ROW)
            {
                NSString *name = [[NSString alloc] initWithUTF8Stri
                 (const char *) sqlite3_column_text(statement, 0)];
                [resultArray addObject:name];
                NSString *department = [[NSString alloc] initWithUT
                (const char *) sqlite3_column_text(statement, 1)];
                [resultArray addObject:department];
                NSString *year = [[NSString alloc]initWithUTF8Strin
                (const char *) sqlite3_column_text(statement, 2)];
                [resultArray addObject:year];
                return resultArray;
            }
            else{
                NSLog(@"Not found");
                return nil;
            }
            sqlite3_reset(statement);
        }
    }
    return nil;
}
```



8、如图所示，更新ViewController.xib文件

9、为上述文本字段创建IBOutlets

10、为上述按钮创建IBAction

11、如下所示，更新ViewController.h

```
#import <UIKit/UIKit.h>
#import "DBManager.h"

@interface ViewController : UIViewController<UITextFieldDelegate>
{
    IBOutlet UITextField *regNoTextField;
    IBOutlet UITextField *nameTextField;
    IBOutlet UITextField *departmentTextField;
    IBOutlet UITextField *yearTextField;
    IBOutlet UITextField *findByRegisterNumberTextField;
    IBOutlet UIScrollView *myScrollView;
}

-(IBAction)saveData:(id)sender;
-(IBAction)findData:(id)sender;

@end
```

12、更新ViewController.m,如下所示

```
#import "ViewController.h"

@interface ViewController ()

@end

@implementation ViewController

- (id)initWithNibName:(NSString *)nibNameOrNil bundle:(NSBundle *)
  nibBundleOrNil
{
    self = [super initWithNibName:nibNameOrNil bundle:nibBundleOrNi
    if (self) {
        // Custom initialization
    }
    return self;
}

- (void)viewDidLoad
{
    [super viewDidLoad];
    // Do any additional setup after loading the view from its nib
}

- (void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}
```

```
-(IBAction)saveData:(id)sender{
    BOOL success = NO;
    NSString *alertString = @"Data Insertion failed";
    if (regNoTextField.text.length>0 &&nameTextField.text.length>0
    departmentTextField.text.length>0 &&yearTextField.text.length>0
    {
        success = [[DBManager getSharedInstance]saveData:
        regNoTextField.text name:nameTextField.text department:
        departmentTextField.text year:yearTextField.text];
    }
    else{
        alertString = @"Enter all fields";
    }
    if (success == NO) {
        UIAlertView *alert = [[UIAlertView alloc]initWithTitle:
        alertString message:nil
        delegate:nil cancelButtonTitle:@"OK" otherButtonTitles:nil]
        [alert show];
    }
}

-(IBAction)findData:(id)sender{
    NSArray *data = [[DBManager getSharedInstance]findByRegisterNum
    findByRegisterNumberTextField.text];
    if (data == nil) {
        UIAlertView *alert = [[UIAlertView alloc]initWithTitle:
        @"Data not found" message:nil delegate:nil cancelButtonTitl
        @"OK" otherButtonTitles:nil];
        [alert show];
        regNoTextField.text = @"";
        nameTextField.text =@"";
        departmentTextField.text = @"";
        yearTextField.text =@"";
    }
    else{
        regNoTextField.text = findByRegisterNumberTextField.text;
        nameTextField.text =[data objectAtIndex:0];
        departmentTextField.text = [data objectAtIndex:1];
        yearTextField.text =[data objectAtIndex:2];
    }
}

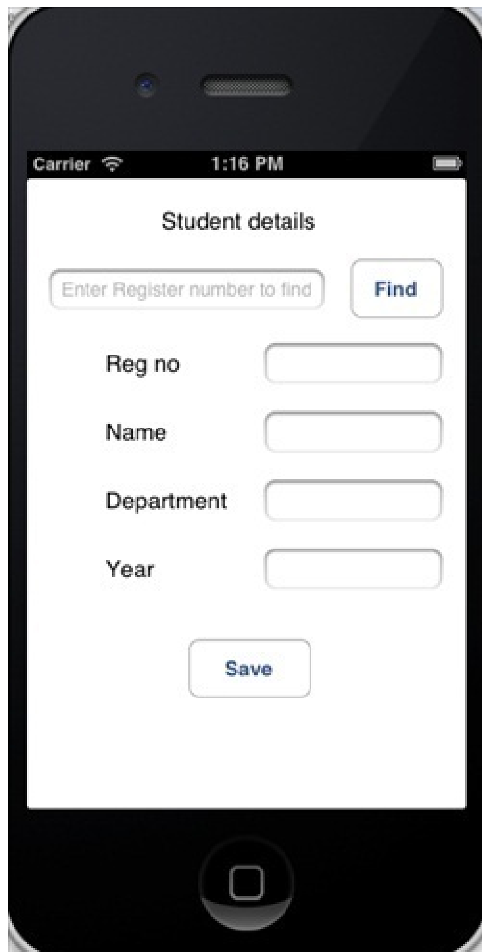
#pragma mark - Text field delegate
-(void)textFieldDidBeginEditing:(UITextField *)textField{
    [myScrollView setFrame:CGRectMake(10, 50, 300, 200)];
    [myScrollView setContentSize:CGSizeMake(300, 350)];
}
-(void)textFieldDidEndEditing:(UITextField *)textField{
    [myScrollView setFrame:CGRectMake(10, 50, 300, 350)];

}
-(BOOL) textFieldShouldReturn:(UITextField *)textField{
```

```
    [textField resignFirstResponder];
    return YES;
}
@end
```

输出

现在当我们运行应用程序时，我们就会获得下面的输出，我们可以在其中添加及查找学生的详细信息



IOS发送电子邮件

简介

我们可以使用IOS设备中的电子邮件应用程序发送电子邮件。

实例步骤

1、创建一个简单的View based application

2、选择项目文件，然后选择目标，然后添加MessageUI.framework

3、在ViewController.xib中添加一个按钮，创建用于发送电子邮件的操作（action）

4、更新ViewController.h,如下所示

```
#import <UIKit/UIKit.h>
#import <MessageUI/MessageUI.h>

@interface ViewController : UIViewController<MFMailComposeViewContr
{
    MFMailComposeViewController *mailComposer;
}

-(IBAction)sendMail:(id)sender;

@end
```

5、如下所示，更新ViewController.m

```
#import "ViewController.h"

@interface ViewController ()

@end

@implementation ViewController

- (void)viewDidLoad
{
    [super viewDidLoad];
}

- (void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}

-(void)sendMail:(id)sender{
    mailComposer = [[MFMailComposeViewController alloc]init];
    mailComposer.mailComposeDelegate = self;
    [mailComposer setSubject:@"Test mail"];
    [mailComposer setMessageBody:@"Testing message
    for the test mail" isHTML:NO];
    [self presentModalViewController:mailComposer animated:YES];
}

#pragma mark - mail compose delegate
-(void)mailComposeController:(MFMailComposeViewController *)contro
 didFinishWithResult:(MFMailComposeResult)result error:(NSError *)e
   if (result) {
        NSLog(@"Result : %d",result);
    }
    if (error) {
        NSLog(@"Error : %@",error);
    }
    [self dismissModalViewControllerAnimated:YES];

}

@end
```

输出

当运行该应用程序，会看如下的输出结果



当点击"send email"发送按钮后，可以看到如下结果：


Carrier
4:41 PM
Cancel
Test mail
Send
To:
Cc/Bcc:
Subject: Test mail
Testing message for the test mail
Sent from my iPhone Simulator

IOS音频和视频(Audio & Video)

简介

音频和视频在最新的设备中颇为常见。

将iosAVFoundation.framework和MediaPlayer.framework添加到Xcode项目中，可以让IOS支持音频和视频(Audio & Video)。

实例步骤

1、创建一个简单的View based application

2、选择项目文件、选择目标，然后添加AVFoundation.framework和MediaPlayer.framework

3、在ViewController.xib中添加两个按钮，创建一个用于分别播放音频和视频的动作（action）

4、更新ViewController.h,如下所示

```
#import <UIKit/UIKit.h>
#import <AVFoundation/AVFoundation.h>
#import <MediaPlayer/MediaPlayer.h>

@interface ViewController : UIViewController
{
    AVAudioPlayer *audioPlayer;
    MPMoviePlayerViewController *moviePlayer;

}
-(IBAction)playAudio:(id)sender;
-(IBAction)playVideo:(id)sender;
@end
```

5、更新ViewController.m，如下所示

```
#import "ViewController.h"

@interface ViewController ()

@end

@implementation ViewController

- (void)viewDidLoad
{
    [super viewDidLoad];
}

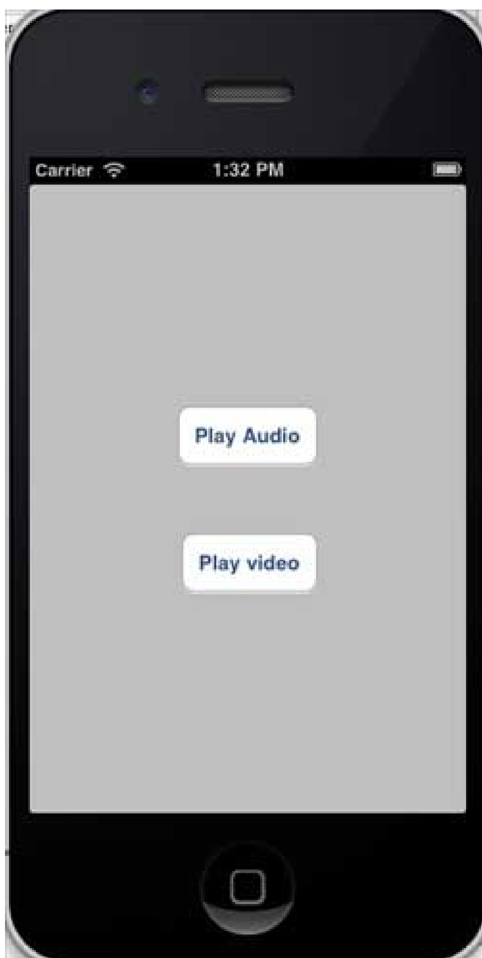
- (void)didReceiveMemoryWarning
{
   [super didReceiveMemoryWarning];
   // Dispose of any resources that can be recreated.
}
-(IBAction)playAudio:(id)sender{
   NSString *path = [[NSBundle mainBundle]
   pathForResource:@"audioTest" ofType:@"mp3"];
   audioPlayer = [[AVAudioPlayer alloc]initWithContentsOfURL:
   [NSURL fileURLWithPath:path] error:NULL];
   [audioPlayer play];
}
-(IBAction)playVideo:(id)sender{
   NSString *path = [[NSBundle mainBundle]pathForResource:
   @"videoTest" ofType:@"mov"];
   moviePlayer = [[MPMoviePlayerViewController
   alloc]initWithContentURL:[NSURL fileURLWithPath:path]];
   [self presentModalViewController:moviePlayer animated:NO];
}
@end
```

注意项

需要添加音频和视频文件，以确保获得预期的输出

输出

运行该程序，得到的输出结果如下



当我们点击 play video(播放视频)显示如下：


Carrier
1:32 PM
Done
0:00
-0:05

IOS文件处理

简介

文件处理不能直观的通过应用程序来解释，我们可以从以下实例来了解IOS的文件处理。

IOS中对文件的操作. 因为应用是在沙箱（sandbox）中的，在文件读写权限上受到限制，只能在几个目录下读写文件。

文件处理中使用的方法

下面列出了用于访问和操作文件的方法的列表。

以下实例你必须替换FilePath1、FilePath和FilePath字符串为完整的文件路径，以获得所需的操作。

检查文件是否存在

```
NSFileManager *fileManager = [NSFileManager defaultManager];
//Get documents directory
NSArray *directoryPaths = NSSearchPathForDirectoriesInDomains
(NSDocumentDirectory, NSUserDomainMask, YES);
NSString *documentsDirectoryPath = [directoryPaths objectAtIndex
if ([fileManager fileExistsAtPath:@""]==YES) {
     NSLog(@"File exists");
 }
```

比较两个文件的内容

```
if ([fileManager contentsEqualAtPath:@"FilePath1" andPath:@" Fil
   NSLog(@"Same content");
}
```

检查是否可写、可读、可执行文件

```
 if ([fileManager isWritableFileAtPath:@"FilePath"]) {
       NSLog(@"isWritable");
  }
  if ([fileManager isReadableFileAtPath:@"FilePath"]) {
       NSLog(@"isReadable");
  }
  if ( [fileManager isExecutableFileAtPath:@"FilePath"]){
       NSLog(@"is Executable");
  }
```

移动文件

```
  if([fileManager moveItemAtPath:@"FilePath1"
  toPath:@"FilePath2" error:NULL]){
       NSLog(@"Moved successfully");
  }
```

复制文件

```
  if ([fileManager copyItemAtPath:@"FilePath1"
  toPath:@"FilePath2"  error:NULL]) {
       NSLog(@"Copied successfully");
  }
```

删除文件

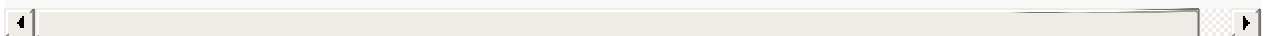
```
if ([fileManager removeItemAtPath:@"FilePath" error:NULL]) {
       NSLog(@"Removed successfully");
  }
```

读取文件

```
NSData *data = [fileManager contentsAtPath:@"Path"];
```

写入文件

```
 [fileManager createFileAtPath:@"" contents:data attributes:nil];
```



IOS地图开发

简介

IOS地图帮助我们定位位置，IOS地图使用 MapKit 框架。

实例步骤

1.创建一个简单的 View based application

2.选择项目文件，然后选择目标，然后添加MapKit.framework.

3.添加 Corelocation.framework

4.向 ViewController.xib 添加地图查看和创建 ibOutlet 并且命名为mapView。

5.通过"File-> New -> File... -> "选择 Objective C class创建一个新的文件，单击下一步

6."sub class of"为 NSObject，类作命名为MapAnnotation

7.选择创建

8.更新MapAnnotation.h，如下所示

```
#import <Foundation/Foundation.h>
#import <MapKit/MapKit.h>

@interface MapAnnotation : NSObject<MKAnnotation>
@property (nonatomic, strong) NSString *title;
@property (nonatomic, readwrite) CLLocationCoordinate2D coordinate;

- (id)initWithTitle:(NSString *)title andCoordinate:
  (CLLocationCoordinate2D)coordinate2d;

@end
```

9.更新MapAnnotation.m，如下所示

```
#import "MapAnnotation.h"

@implementation MapAnnotation
-(id)initWithTitle:(NSString *)title andCoordinate:
 (CLLocationCoordinate2D)coordinate2d{
    self.title = title;
    self.coordinate =coordinate2d;
    return self;
}
@end
```

10.更新ViewController.h， 如下所示

```
#import <UIKit/UIKit.h>
#import <MapKit/MapKit.h>
#import <CoreLocation/CoreLocation.h>
@interface ViewController : UIViewController<MKMapViewDelegate>
{
    MKMapView *mapView;
}
@end
```

11.更新ViewController.m， 如下所示

```
#import "ViewController.h"
#import "MapAnnotation.h"

@interface ViewController ()

@end

@implementation ViewController

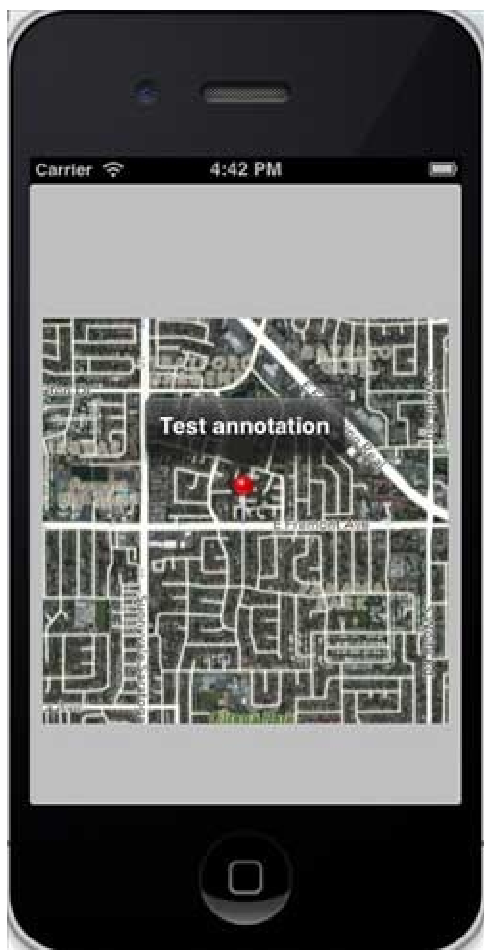
- (void)viewDidLoad
{
   [super viewDidLoad];
   mapView = [[MKMapView alloc]initWithFrame:
   CGRectMake(10, 100, 300, 300)];
   mapView.delegate = self;
   mapView.centerCoordinate = CLLocationCoordinate2DMake(37.32, -12
   mapView.mapType = MKMapTypeHybrid;
   CLLocationCoordinate2D location;
   location.latitude = (double) 37.332768;
   location.longitude = (double) -122.030039;
   // Add the annotation to our map view
   MapAnnotation *newAnnotation = [[MapAnnotation alloc]
   initWithTitle:@"Apple Head quaters" andCoordinate:location];
   [mapView addAnnotation:newAnnotation];
   CLLocationCoordinate2D location2;
   location2.latitude = (double) 37.35239;
   location2.longitude = (double) -122.025919;
   MapAnnotation *newAnnotation2 = [[MapAnnotation alloc]
   initWithTitle:@"Test annotation" andCoordinate:location2];
   [mapView addAnnotation:newAnnotation2];
   [self.view addSubview:mapView];
}
// When a map annotation point is added, zoom to it (1500 range)
- (void)mapView:(MKMapView *)mv didAddAnnotationViews:(NSArray *)vi
{
   MKAnnotationView *annotationView = [views objectAtIndex:0];
   id <MKAnnotation> mp = [annotationView annotation];
   MKCoordinateRegion region = MKCoordinateRegionMakeWithDistance
   ([mp coordinate], 1500, 1500);
   [mv setRegion:region animated:YES];
   [mv selectAnnotation:mp animated:YES];
}

- (void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}

@end
```

输出

运行应用程序时，输出结果如下



当我们向上滚动地图时，输出结果如下


Carrier
4:42 PM
Apple Head quaters

IOS应用内购买

简介

应用程序内购买是应用程序用于购买额外内容或升级功能。

实例步骤

1.在 iTunes 连接中请确保拥有一个唯一的 App ID（unique App ID ），当创建捆绑的ID（ bundle ID）应用程序更新时，代码会以相应的配置文件签名在Xcode上

2.创建新的应用程序和更新应用程序信息。你可以知道更多有关的，在苹果的 添加新的应用程序 文档中

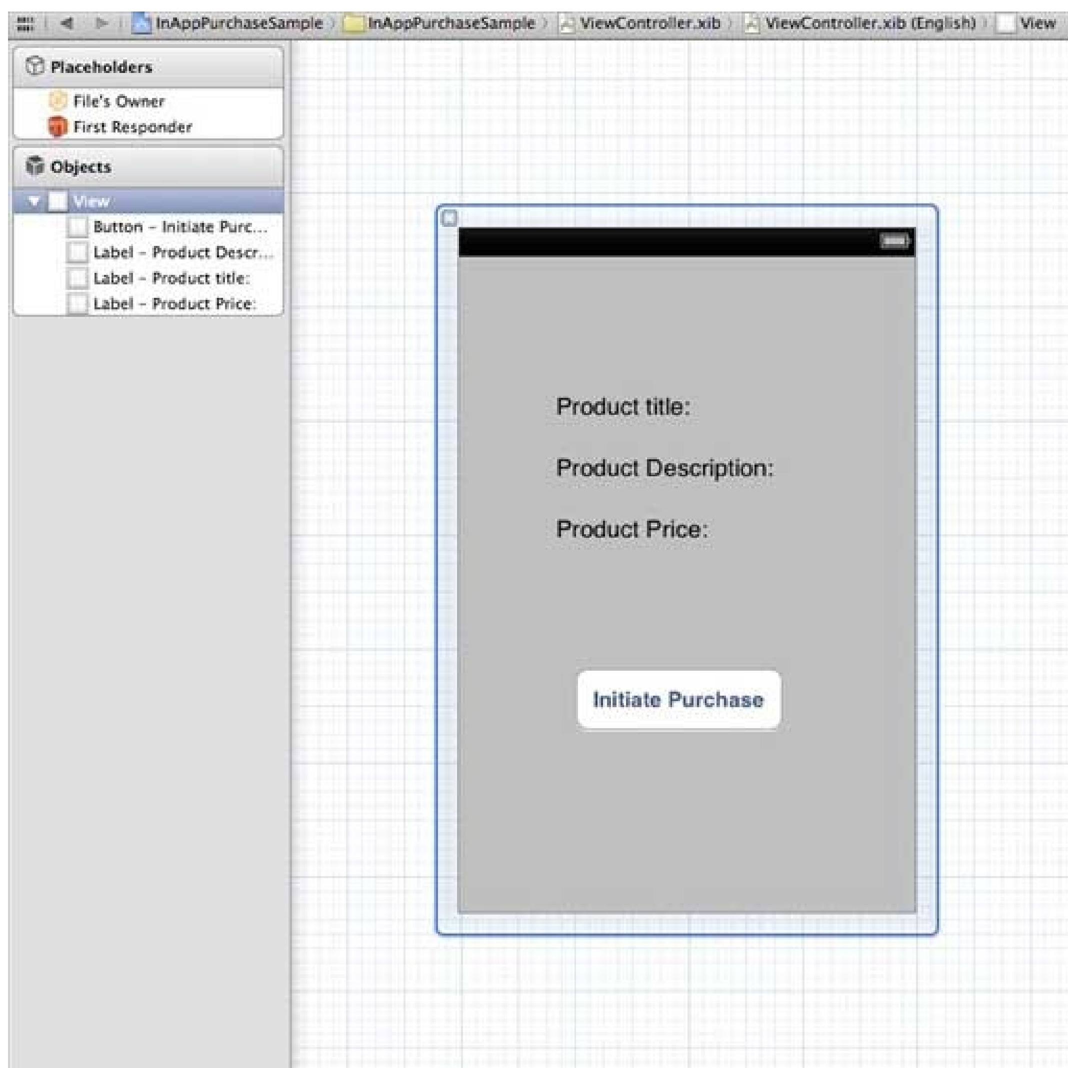
3.在应用程序页的管理应用程序(Manage In-App Purchase)中，为app内付费添加新产品

4.确保设置的应用程序为的银行详细。需要将其设置为在应用程序内购买（In-App purchase）。此外在 iTunes 中使用管理用户（Manage Users）选项，创建一个测试用户帐户连接您的应用程序的页。

5.下一步是与处理代码和为我们在应用程序内购买创建有关的 UI。

6.创建一个单一的视图应用程序，并在 iTunes 中指定的标识符连接输入捆绑标识符

7.更新ViewController.xib ，如下所示



8.为三个标签创建IBOutlets，且将按钮分别命名为 productTitleLabel、productDescriptionLabel、 productPriceLabel 和 purchaseButton

9.选择项目文件，然后选择目标，然后添加StoreKit.framework

10.更新ViewController.h ， 如下所示

```
#import <UIKit/UIKit.h>
#import <StoreKit/StoreKit.h>

@interface ViewController : UIViewController<
SKProductsRequestDelegate,SKPaymentTransactionObserver>
{
    SKProductsRequest *productsRequest;
    NSArray *validProducts;
    UIActivityIndicatorView *activityIndicatorView;
    IBOutlet UILabel *productTitleLabel;
    IBOutlet UILabel *productDescriptionLabel;
    IBOutlet UILabel *productPriceLabel;
    IBOutlet UIButton *purchaseButton;
}
- (void)fetchAvailableProducts;
- (BOOL)canMakePurchases;
- (void)purchaseMyProduct:(SKProduct*)product;
- (IBAction)purchase:(id)sender;

@end
```

11.更新ViewController.m， 如下所示

```
#import "ViewController.h"
#define kTutorialPointProductID
@"com.tutorialPoints.testApp.testProduct"

@interface ViewController ()

@end

@implementation ViewController

- (void)viewDidLoad
{
    [super viewDidLoad];
    // Adding activity indicator
    activityIndicatorView = [[UIActivityIndicatorView alloc]
    initWithActivityIndicatorStyle:UIActivityIndicatorViewStyleWhit
    activityIndicatorView.center = self.view.center;
    [activityIndicatorView hidesWhenStopped];
    [self.view addSubview:activityIndicatorView];
    [activityIndicatorView startAnimating];
    //Hide purchase button initially
    purchaseButton.hidden = YES;
    [self fetchAvailableProducts];
}

- (void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
```

```
    // Dispose of any resources that can be recreated.
}

-(void)fetchAvailableProducts{
    NSSet *productIdentifiers = [NSSet
    setWithObjects:kTutorialPointProductID,nil];
    productsRequest = [[SKProductsRequest alloc]
    initWithProductIdentifiers:productIdentifiers];
    productsRequest.delegate = self;
    [productsRequest start];
}

- (BOOL)canMakePurchases
{
    return [SKPaymentQueue canMakePayments];
}
- (void)purchaseMyProduct:(SKProduct*)product{
    if ([self canMakePurchases]) {
        SKPayment *payment = [SKPayment paymentWithProduct:product]
        [[SKPaymentQueue defaultQueue] addTransactionObserver:self]
        [[SKPaymentQueue defaultQueue] addPayment:payment];
    }
    else{
        UIAlertView *alertView = [[UIAlertView alloc]initWithTitle:
        @"Purchases are disabled in your device" message:nil delega
        self cancelButtonTitle:@"Ok" otherButtonTitles: nil];
        [alertView show];
    }
}
-(IBAction)purchase:(id)sender{
    [self purchaseMyProduct:[validProducts objectAtIndex:0]];
    purchaseButton.enabled = NO;
}

#pragma mark StoreKit Delegate

-(void)paymentQueue:(SKPaymentQueue *)queue
 updatedTransactions:(NSArray *)transactions {
    for (SKPaymentTransaction *transaction in transactions) {
        switch (transaction.transactionState) {
            case SKPaymentTransactionStatePurchasing:
                    NSLog(@"Purchasing");
                break;
            case SKPaymentTransactionStatePurchased:
               if ([transaction.payment.productIdentifier
                  isEqualToString:kTutorialPointProductID]) {
                   NSLog(@"Purchased ");
                   UIAlertView *alertView = [[UIAlertView alloc]ini
                   @"Purchase is completed succesfully" message:nil
                   self cancelButtonTitle:@"Ok" otherButtonTitles:
                   [alertView show];
                }
                [[SKPaymentQueue defaultQueue] finishTransaction:tr
```

```
                break;
            case SKPaymentTransactionStateRestored:
                NSLog(@"Restored ");
                [[SKPaymentQueue defaultQueue] finishTransaction:tr
                break;
            case SKPaymentTransactionStateFailed:
                NSLog(@"Purchase failed ");
                break;
            default:
                break;
        }
    }
}

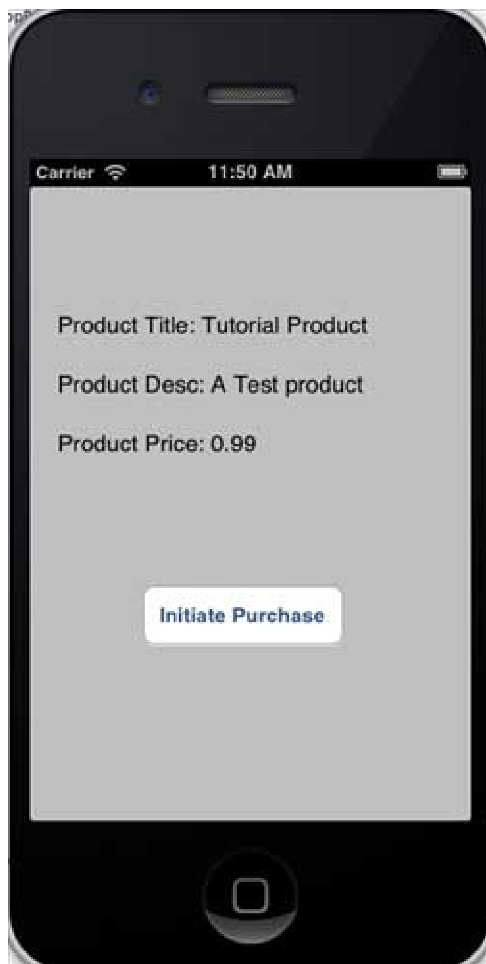
-(void)productsRequest:(SKProductsRequest *)request
 didReceiveResponse:(SKProductsResponse *)response
{
    SKProduct *validProduct = nil;
    int count = [response.products count];
    if (count>0) {
        validProducts = response.products;
        validProduct = [response.products objectAtIndex:0];
        if ([validProduct.productIdentifier
           isEqualToString:kTutorialPointProductID]) {
            [productTitleLabel setText:[NSString stringWithFormat:
            @"Product Title: %@",validProduct.localizedTitle]];
            [productDescriptionLabel setText:[NSString stringWithFo
            @"Product Desc: %@",validProduct.localizedDescription]]
            [productPriceLabel setText:[NSString stringWithFormat:
            @"Product Price: %@",validProduct.price]];
        }
    } else {
        UIAlertView *tmp = [[UIAlertView alloc]
                            initWithTitle:@"Not Available"
                            message:@"No products to purchase"
                            delegate:self
                            cancelButtonTitle:nil
                            otherButtonTitles:@"Ok", nil];
        [tmp show];
    }
    [activityIndicatorView stopAnimating];
    purchaseButton.hidden = NO;
}

@end
```

注意： 需要修改你创建In-App Pur（应用内购买）的 kTutorialPointProductID 。通过修改fetchAvailableProducts产品标识符的 NSSet, 你可以添加多个产品。

输出

运行该应用程序,输出结果如下



确保已经中登录。单击购买选择现有的Apple ID。输入有效的测试帐户的用户名和密码。几秒钟后，显示下面的信息



一旦产品成功购买，将获得以下信息。可以在显示此信息的地方，更新应用功能相关的代码


Carrier
11:55 AM
Product Title: Tutorial Product
Product Desc: A Test product
Purchase is completed succesfully
Ok
Initiate Purchase

IOS iAD整合

简介

IAD是苹果推出的广告平台，它可以帮助开发者从应用程序中获取收入。

实例步骤

1. 创建一个简单的View based application

2. 选择项目文件，然后选择目标，然后选择框架并添加 iAd.framework。

3. 更新 ViewController.h 如下所示

```
#import <UIKit/UIKit.h>
#import <iAd/iAd.h>
@interface ViewController : UIViewController<ADBannerViewDelegate>
{
    ADBannerView *bannerView;
}
@end
```

4. 更新ViewController.m， 如下所示

```
#import "ViewController.h"

@interface ViewController ()

@end

@implementation ViewController

- (void)viewDidLoad
{
    [super viewDidLoad];
    bannerView = [[ADBannerView alloc]initWithFrame:
    CGRectMake(0, 0, 320, 50)];
    // Optional to set background color to clear color
    [bannerView setBackgroundColor:[UIColor clearColor]];
    [self.view addSubview: bannerView];
}

- (void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}

#pragma mark - AdViewDelegates

-(void)bannerView:(ADBannerView *)banner
 didFailToReceiveAdWithError:(NSError *)error{
    NSLog(@"Error loading");
}

-(void)bannerViewDidLoadAd:(ADBannerView *)banner{
    NSLog(@"Ad loaded");
}
-(void)bannerViewWillLoadAd:(ADBannerView *)banner{
    NSLog(@"Ad will load");
}
-(void)bannerViewActionDidFinish:(ADBannerView *)banner{
    NSLog(@"Ad did finish");

}
@end
```

输出

运行该应用程序,得到如下输出结果:


Carrier
4:28 PM
$1234
Easily generate more revenue from your application.
Find Out How
iAd

IOS GameKit

简介

GameKit是iOS SDK中一个常用的框架。其核心功能有3个：

- 交互游戏平台Game Center，
- P2P设备通讯功能
- In-Game Voice。

实例步骤

1.在链接 iTunes 时请确保拥有一个唯一的 App ID（ unique App ID），App ID在我们应用程序更新 bundle ID时及在Xcode代码签名与相应的配置文件需要使用到。

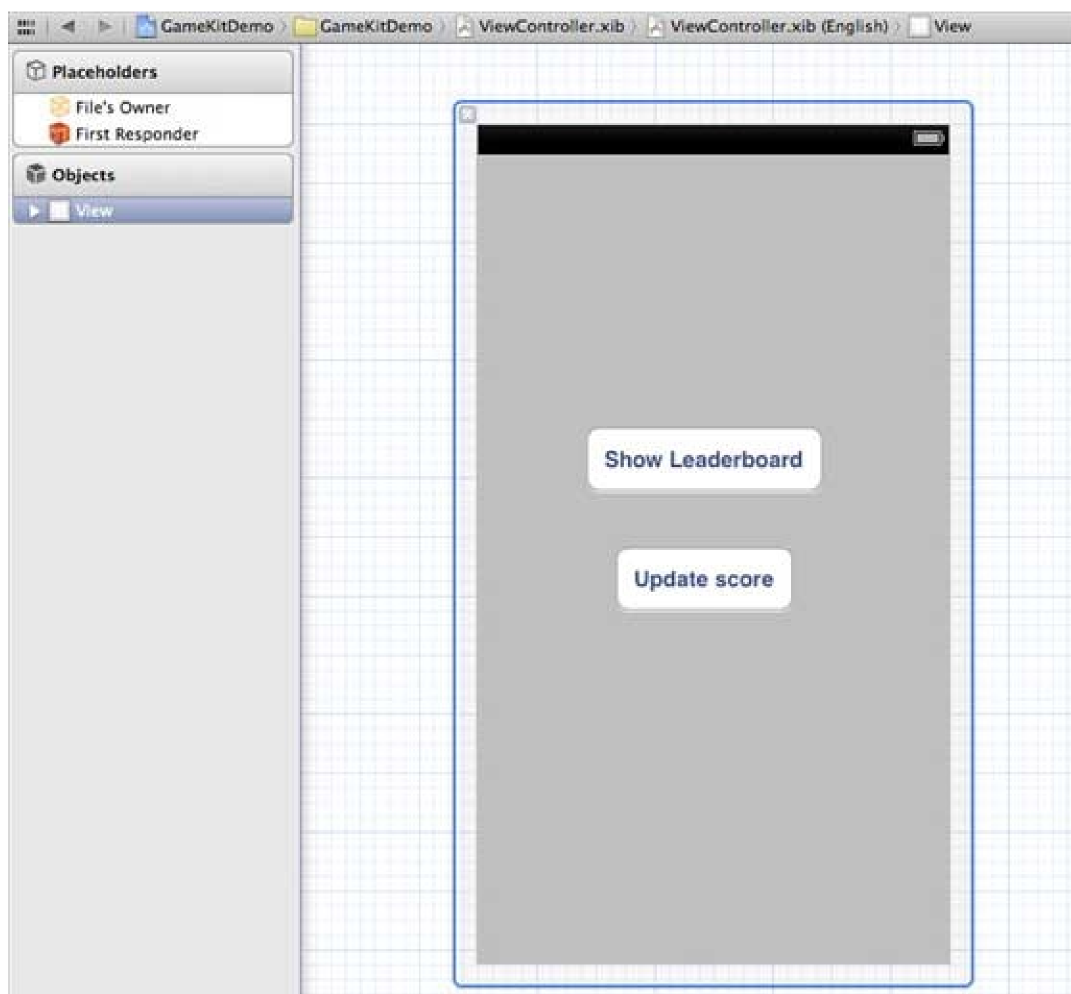
2.创建新的应用程序和更新应用程序信息。在添加新的应用程序文档可以了解更多有关信息。

3.打开你申请的application,点击Manage Game Center选项。进入后点击Enable Game Center使你的Game Center生效。接下来设置自己的Leaderboard和Achievements。

4.下一步涉及处理代码，并为我们的应用程序创建用户界面。

5.创建一个single view application，并输入 bundle identifier 。

6.更新 ViewController.xib，如下所示



7.选择项目文件，然后选择目标，然后添加GameKit.framework

8.为已添加的按钮创建IBActions

9.更新ViewController.h文件，如下所示

```
#import <UIKit/UIKit.h>
#import <GameKit/GameKit.h>

@interface ViewController : UIViewController
<GKLeaderboardViewControllerDelegate>

-(IBAction)updateScore:(id)sender;
-(IBAction)showLeaderBoard:(id)sender;

@end
```

10.更新ViewController.m， 如下所示

```
#import "ViewController.h"

@interface ViewController ()
```

```
@end

@implementation ViewController

- (void)viewDidLoad
{
    [super viewDidLoad];
    if([GKLocalPlayer localPlayer].authenticated == NO)
    {
      [[GKLocalPlayer localPlayer]
      authenticateWithCompletionHandler:^(NSError *error)
      {
         NSLog(@"Error%@",error);
      }];
    }
}

- (void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}
- (void) updateScore: (int64_t) score
forLeaderboardID: (NSString*) category
{
    GKScore *scoreObj = [[GKScore alloc]
    initWithCategory:category];
    scoreObj.value = score;
    scoreObj.context = 0;
    [scoreObj reportScoreWithCompletionHandler:^(NSError *error) {
        // Completion code can be added here
        UIAlertView *alert = [[UIAlertView alloc]
        initWithTitle:nil message:@"Score Updated Succesfully"
        delegate:self cancelButtonTitle:@"Ok" otherButtonTitles: ni
        [alert show];

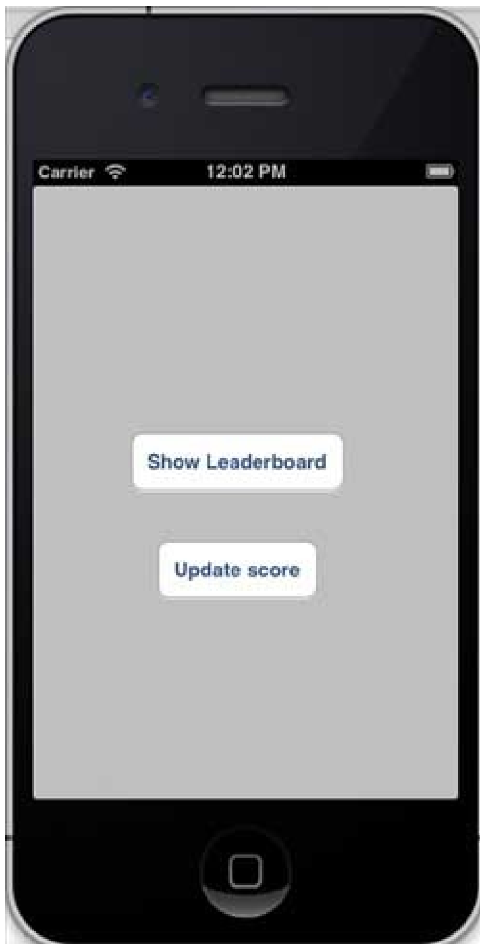
    }];
}
-(IBAction)updateScore:(id)sender{
    [self updateScore:200 forLeaderboardID:@"tutorialsPoint"];
}
-(IBAction)showLeaderBoard:(id)sender{
    GKLeaderboardViewController *leaderboardViewController =
    [[GKLeaderboardViewController alloc] init];
    leaderboardViewController.leaderboardDelegate = self;
    [self presentModalViewController:
    leaderboardViewController animated:YES];

}
#pragma mark - Gamekit delegates
- (void)leaderboardViewControllerDidFinish:
(GKLeaderboardViewController *)viewController{
    [self dismissModalViewControllerAnimated:YES];
```

```
}
@end
```

输出

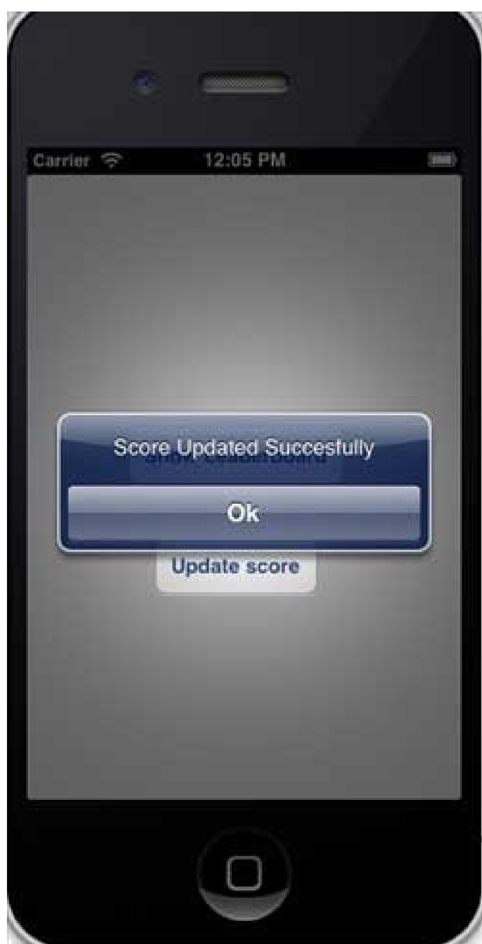
运行该应用程序，输出结果如下



当我们单击显示排行榜时，屏幕显示如下:



当我们点击更新分数，比分将被更新到我们排行榜上，我们会得到一个信息，如下图所示


Carrier
12:05 PM
Score Updated Succesfully
Ok
Update score

IOS 故事板(Storyboards)

简介

Storyboards在 iOS 5 中才有介绍，当我们用Storyboards时，部署目标应该是iOS5.0或更高版本。

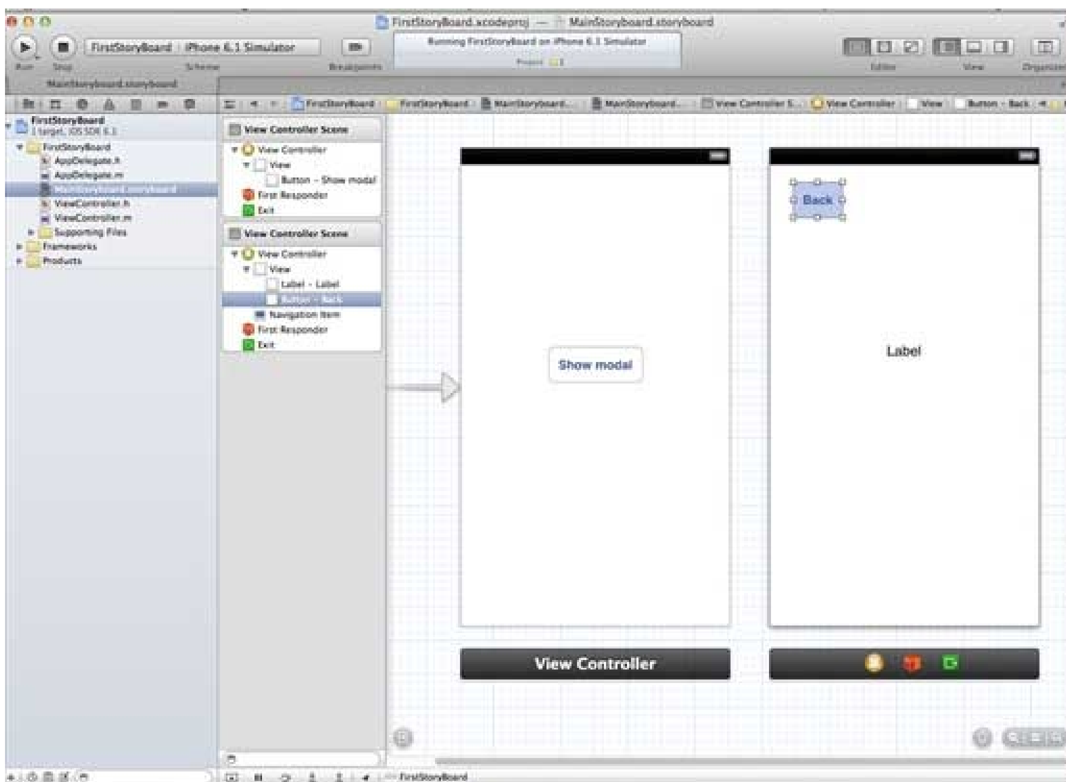
Storyboards 帮助我们了解视觉流动的画面，在界面为

MainStoryboard.storyboard下创建所有应用程序屏幕。

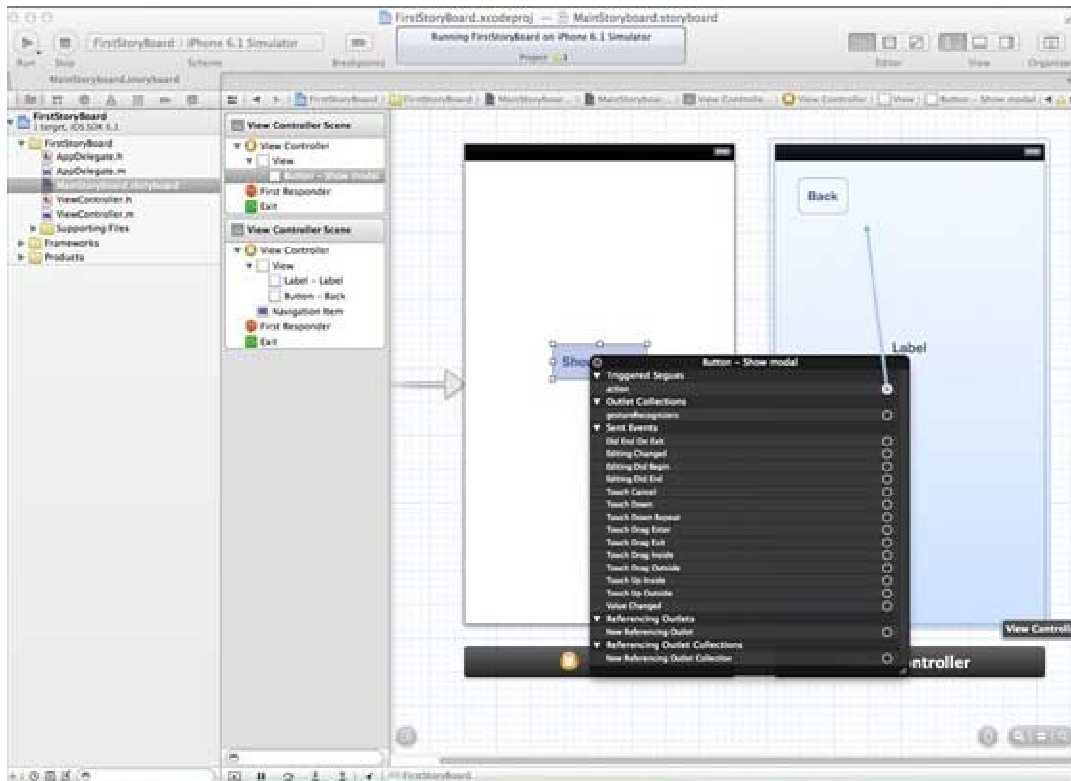
实例步骤

1. 创建一个single view application，创建应用程序时选择 storyboard 复选框。

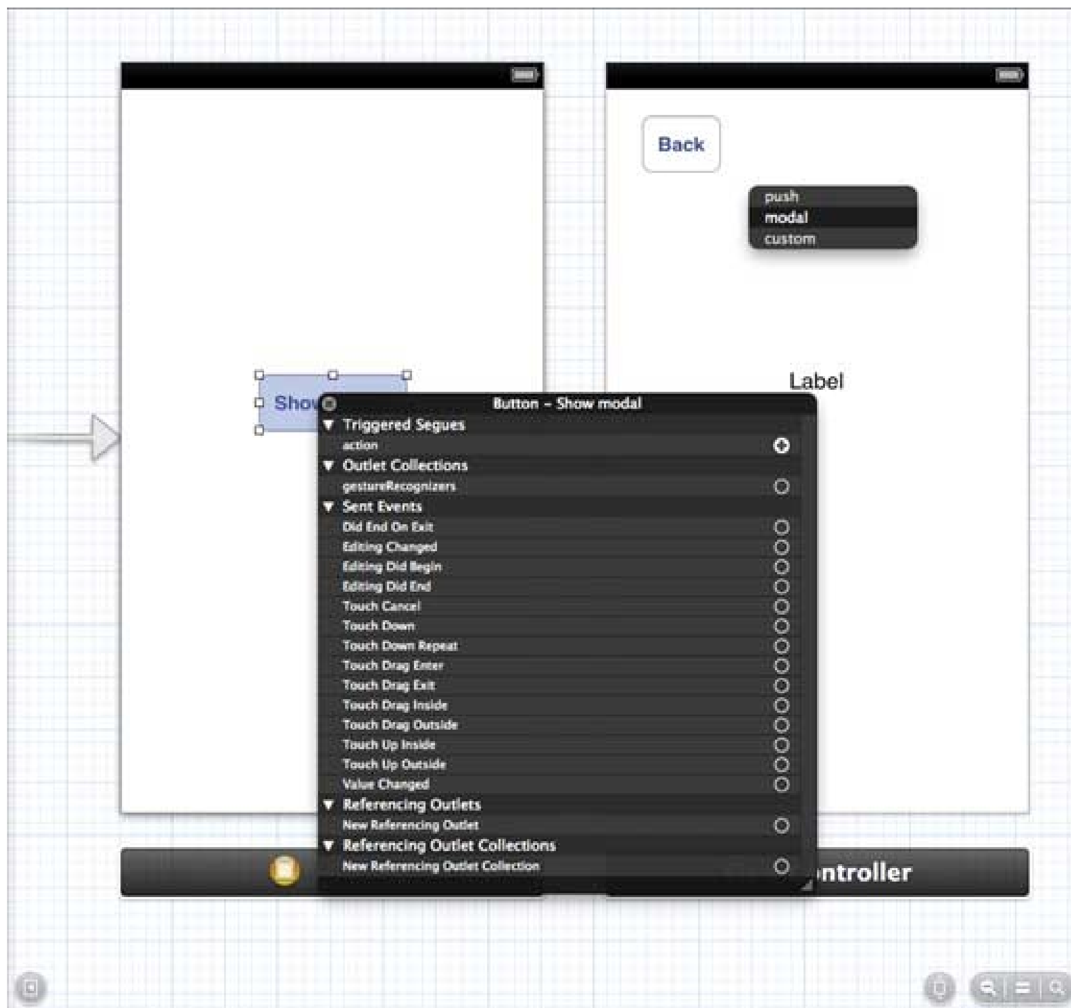
2. 选择MainStoryboard.storyboard，在这里你可以找到单一视图控制器。添加一个视图控制器，更新视图控制器，如下所示



3.连接两个视图控制器。右键单击"show modal（显示模式）"按钮，在左侧视图控制器将其拖动到右视视图控制器中,如下图所示：



4.现在从如下所示的三个显示选项中选择modal(模态)



5.更新 ViewController.h 如下所示

```
#import <UIKit/UIKit.h>

@interface ViewController : UIViewController

-(IBAction)done:(UIStoryboardSegue *)seque;

@end
```

6.更新 ViewController.m 如下所示

```
#import "ViewController.h"

@interface ViewController ()

@end

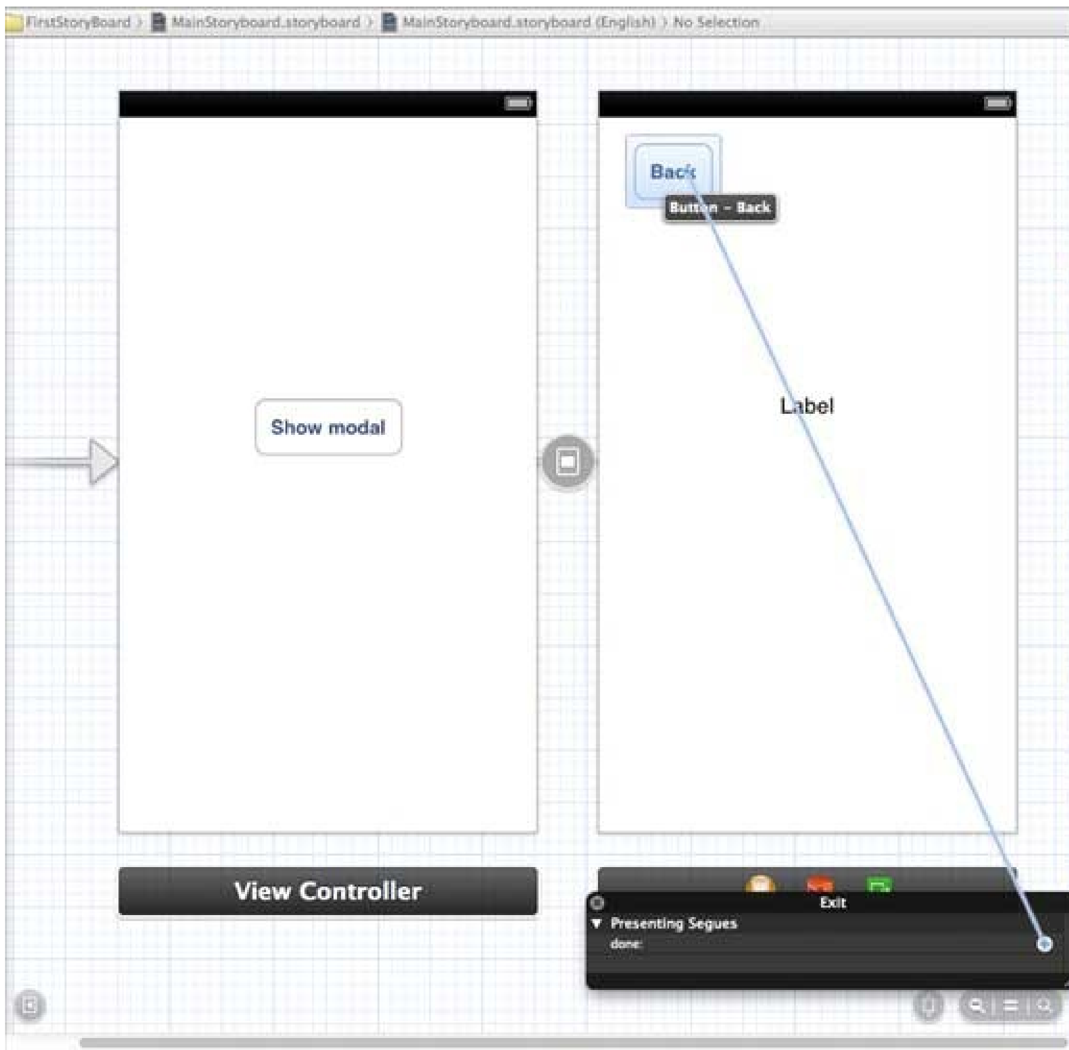
@implementation ViewController

- (void)viewDidLoad
{
    [super viewDidLoad];
}

- (void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}
-(IBAction)done:(UIStoryboardSegue *)seque{
    [self.navigationController popViewControllerAnimated:YES];
}

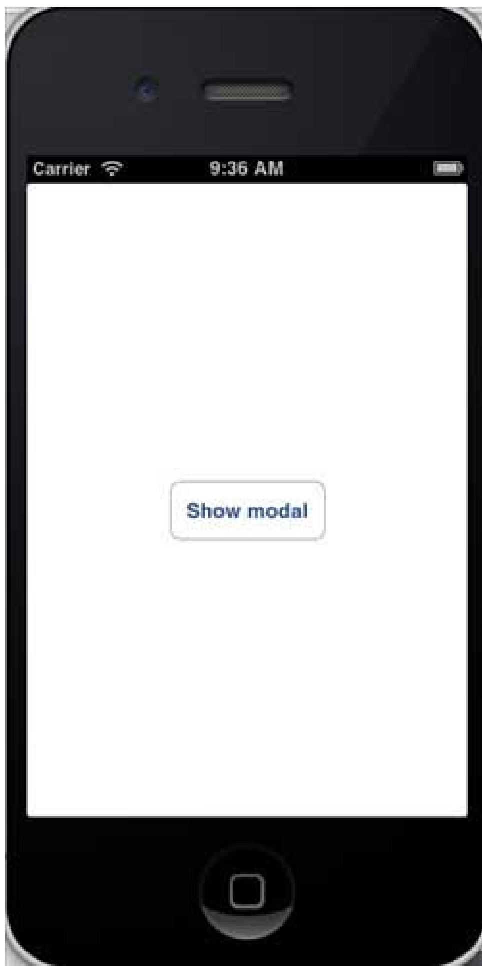
@end
```

7.选择"MainStoryboard.storyboard"，并右键点击"Exit "按钮，在右侧视图控制器中选择和连接后退按钮，如下图所示



输出

在iPhone设备中运行该应用程序,得到如下输出结果



现在，选择显示模式，将得到下面的输出结果


Carrier
9:36 AM
Back
Label

IOS自动布局

简介

自动布局在iOS 6.0中引入，仅可以支持IOS6.0 及 更高版本。它可以帮助我们创建用于多个种设备的界面。

实例步骤

1.创建一个简单的 View based application

2.修改 ViewController.m 的文件内容，如下所示

```
#import "ViewController.h"
@interface ViewController ()
@property (nonatomic, strong) UIButton *leftButton;
@property (nonatomic, strong) UIButton *rightButton;
@property (nonatomic, strong) UITextField *textfield;

@end
@implementation ViewController

- (void)viewDidLoad{
    [super viewDidLoad];
    UIView *superview = self.view;
    /*1\. Create leftButton and add to our view*/
    self.leftButton = [UIButton buttonWithType:UIButtonTypeRoundedR
    self.leftButton.translatesAutoresizingMaskIntoConstraints = NO;
    [self.leftButton setTitle:@"LeftButton" forState:UIControlState
    [self.view addSubview:self.leftButton];
    /* 2\. Constraint to position LeftButton's X*/
    NSLayoutConstraint *leftButtonXConstraint = [NSLayoutConstraint
    constraintWithItem:self.leftButton attribute:NSLayoutAttributeC
    relatedBy:NSLayoutRelationGreaterThanOrEqual toItem:superview a
    NSLayoutAttributeCenterX multiplier:1.0 constant:-60.0f];
    /* 3\. Constraint to position LeftButton's Y*/
    NSLayoutConstraint *leftButtonYConstraint = [NSLayoutConstraint
    constraintWithItem:self.leftButton attribute:NSLayoutAttributeC
    relatedBy:NSLayoutRelationEqual toItem:superview attribute:
    NSLayoutAttributeCenterY multiplier:1.0f constant:0.0f];
    /* 4\. Add the constraints to button's superview*/
    [superview addConstraints:@[ leftButtonXConstraint,
    leftButtonYConstraint]];
    /*5\. Create rightButton and add to our view*/
    self.rightButton = [UIButton buttonWithType:UIButtonTypeRounded
    self.rightButton.translatesAutoresizingMaskIntoConstraints = NO
    [self.rightButton setTitle:@"RightButton" forState:UIControlSta
```

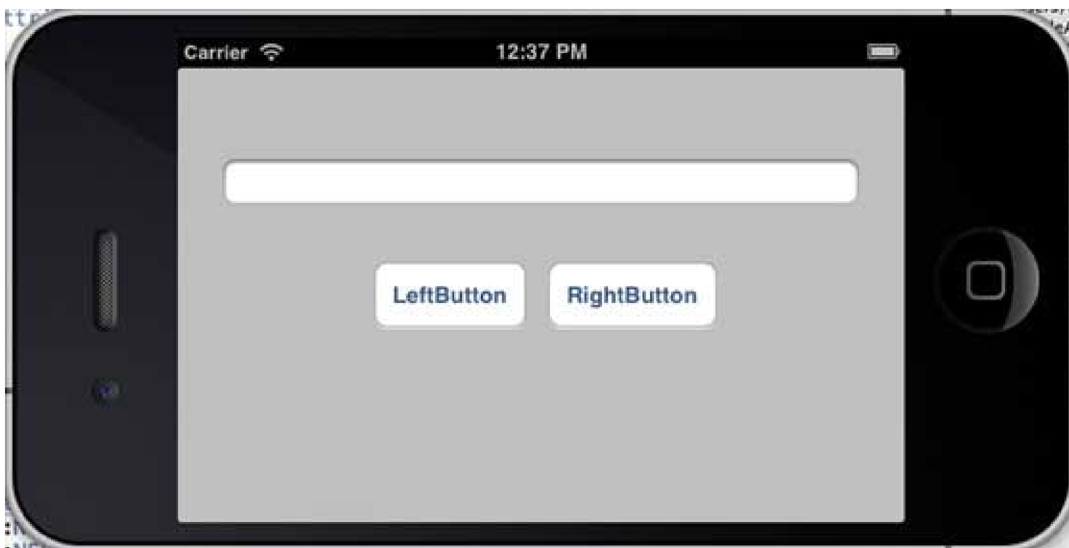
```
    [self.view addSubview:self.rightButton];
    /*6\. Constraint to position RightButton's X*/
    NSLayoutConstraint *rightButtonXConstraint = [NSLayoutConstrair
    constraintWithItem:self.rightButton attribute:NSLayoutAttribute
    relatedBy:NSLayoutRelationGreaterThanOrEqual toItem:superview a
    NSLayoutAttributeCenterX multiplier:1.0 constant:60.0f];
    /*7\. Constraint to position RightButton's Y*/
    rightButtonXConstraint.priority = UILayoutPriorityDefaultHigh;
    NSLayoutConstraint *centerYMyConstraint = [NSLayoutConstraint
    constraintWithItem:self.rightButton attribute:NSLayoutAttribute
    relatedBy:NSLayoutRelationGreaterThanOrEqual toItem:superview a
    NSLayoutAttributeCenterY multiplier:1.0f constant:0.0f];
    [superview addConstraints:@[centerYMyConstraint,
    rightButtonXConstraint]];
   //8\. Add Text field
    self.textfield = [[UITextField alloc]initWithFrame:
    CGRectMake(0, 100, 100, 30)];
    self.textfield.borderStyle = UITextBorderStyleRoundedRect;
    self.textfield.translatesAutoresizingMaskIntoConstraints = NO;
    [self.view addSubview:self.textfield];
    //9\. Text field Constraints
    NSLayoutConstraint *textFieldTopConstraint = [NSLayoutConstrair
    constraintWithItem:self.textfield attribute:NSLayoutAttributeTo
    relatedBy:NSLayoutRelationGreaterThanOrEqual toItem:superview
    attribute:NSLayoutAttributeTop multiplier:1.0 constant:60.0f];
    NSLayoutConstraint *textFieldBottomConstraint = [NSLayoutConstr
    constraintWithItem:self.textfield attribute:NSLayoutAttributeTo
    relatedBy:NSLayoutRelationGreaterThanOrEqual toItem:self.rightE
    attribute:NSLayoutAttributeTop multiplier:0.8 constant:-60.0f];
    NSLayoutConstraint *textFieldLeftConstraint = [NSLayoutConstrai
    constraintWithItem:self.textfield attribute:NSLayoutAttributeLe
    relatedBy:NSLayoutRelationEqual toItem:superview attribute:
    NSLayoutAttributeLeft multiplier:1.0 constant:30.0f];
    NSLayoutConstraint *textFieldRightConstraint = [NSLayoutConstra
    constraintWithItem:self.textfield attribute:NSLayoutAttributeRi
    relatedBy:NSLayoutRelationEqual toItem:superview attribute:
    NSLayoutAttributeRight multiplier:1.0 constant:-30.0f];
    [superview addConstraints:@[textFieldBottomConstraint ,
    textFieldLeftConstraint, textFieldRightConstraint,
    textFieldTopConstraint]];
}
- (void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}
@end
```

输出

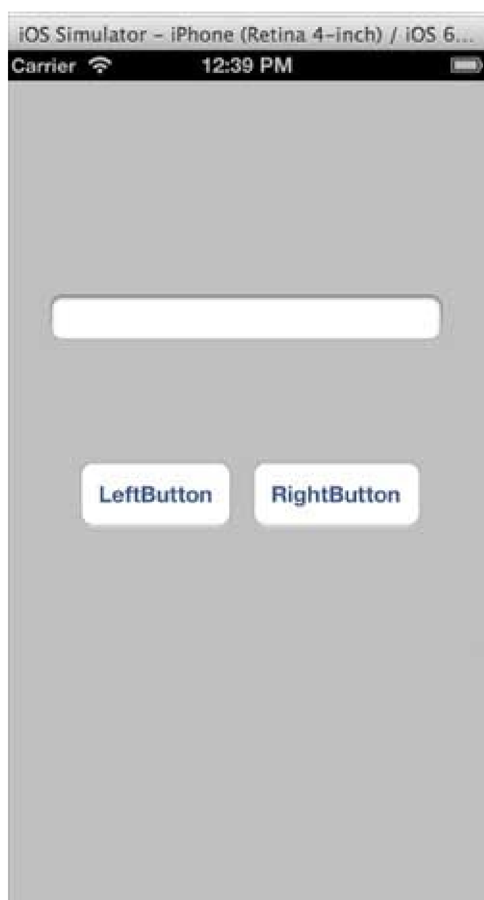
运行应用程序，在 iPhone 模拟器上会有下面的输出结果



当我们更改模拟器为横向的方向时，输出结果如下



我们在 iPhone 5 模拟器上运行同一应用程序时,输出结果如下



当我们更改模拟器为横向的方向时，输出结果如下:



IOS-Twitter和Facebook

简介

Twitter已经整合到iOS5.0，而Facebook已经被集成在 iOS 6.0中。本教程的重点讲解如何利用苹果提供的类在iOS5.0和iOS6.0中部署Twitter和Facebook。

实例步骤

1. 创建一个简单View based application

2. 选择项目文件，然后选择"targets(目标)"，然后在 choose frameworks（选择框架）中添加Social.framework 和 Accounts.framework

3. 添加两个名为facebookPost 和 twitterPost的按钮，并为他们创建 ibActions。

4. 更新 ViewController.h 如下

```
#import <Social/Social.h>
#import <Accounts/Accounts.h>
#import <UIKit/UIKit.h>

@interface ViewController : UIViewController

-(IBAction)twitterPost:(id)sender;
-(IBAction)facebookPost:(id)sender;

@end
```

5. 更新ViewController.m， 如下所示

```
#import "ViewController.h"

@interface ViewController ()

@end

@implementation ViewController

- (void)viewDidLoad
{
    [super viewDidLoad];
}

- (void)didReceiveMemoryWarning
{
```

```
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}

-(IBAction)facebookPost:(id)sender{

   SLComposeViewController *controller = [SLComposeViewController
   composeViewControllerForServiceType:SLServiceTypeFacebook];
   SLComposeViewControllerCompletionHandler myBlock =
   ^(SLComposeViewControllerResult result){
          if (result == SLComposeViewControllerResultCancelled)
          {
            NSLog(@"Cancelled");
          }
          else
          {
            NSLog(@"Done");
          }
          [controller dismissViewControllerAnimated:YES completion:
        };
   controller.completionHandler =myBlock;
   //Adding the Text to the facebook post value from iOS
   [controller setInitialText:@"My test post"];
   //Adding the URL to the facebook post value from iOS
   [controller addURL:[NSURL URLWithString:@"http://www.test.com"]]
   //Adding the Text to the facebook post value from iOS
   [self presentViewController:controller animated:YES completion:n
}

-(IBAction)twitterPost:(id)sender{
    SLComposeViewController *tweetSheet = [SLComposeViewController
    composeViewControllerForServiceType:SLServiceTypeTwitter];
    [tweetSheet setInitialText:@"My test tweet"];
    [self presentModalViewController:tweetSheet animated:YES];
}

@end
```

输出

运行该应用程序并单击 facebookPost 时我们将获得以下输出



当我们单击 twitterPost 时，我们将获得以下输出


Carrier
4:59 PM
Cancel
Tweet
Send
My test tweet
Add Location
127
Q W E R T Y U I O P
A S D F G H J K L
Z X C V B N M
.?123
space
@
#

IOS内存管理

简介

iOS下内存管理的基本思想就是引用计数，通过对象的引用计数来对内存对象的生命周期进行控制。具体到编程时间方面，主要有两种方式：

1：MRR（manual retain-release），人工引用计数，对象的生成、销毁、引用计数的变化都是由开发人员来完成。

2：ARC（Automatic Reference Counting），自动引用计数，只负责对象的生成，其他过程开发人员不再需要关心其销毁，使用方式类似于垃圾回收，但其实质还是引用计数。

面临的问题

根据苹果说明文档，面临的两个主要问题是：

释放或覆盖的数据仍然在使用。这将造成内存损坏，通常在应用程序崩溃，或者更糟，损坏用户数据。

不释放不再使用的数据会导致内存泄漏。分配的内存，内存泄漏不会释放，即使它从来没有再次使用。泄漏会导致应用程序的内存使用量日益增加，这反过来又可能会导致系统性能较差或死机。

内存管理规则

我们创建自己的对象，当他们不再需要的时候，释放他们。

保留需要使用的对象。如果没有必要必须释放这些对象。

不要释放我们没有拥有的对象。

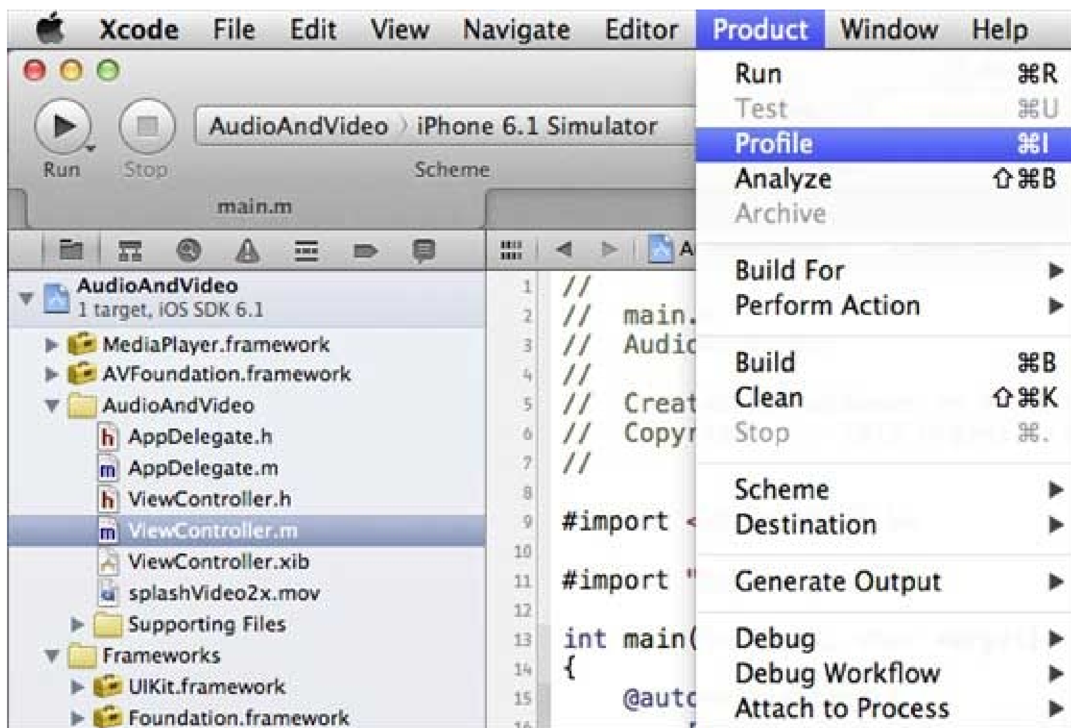
使用内存管理工具

可以用Xcode工具仪器的帮助下分析内存的使用情况。它包括的工具有活动监视器，分配，泄漏，僵尸等

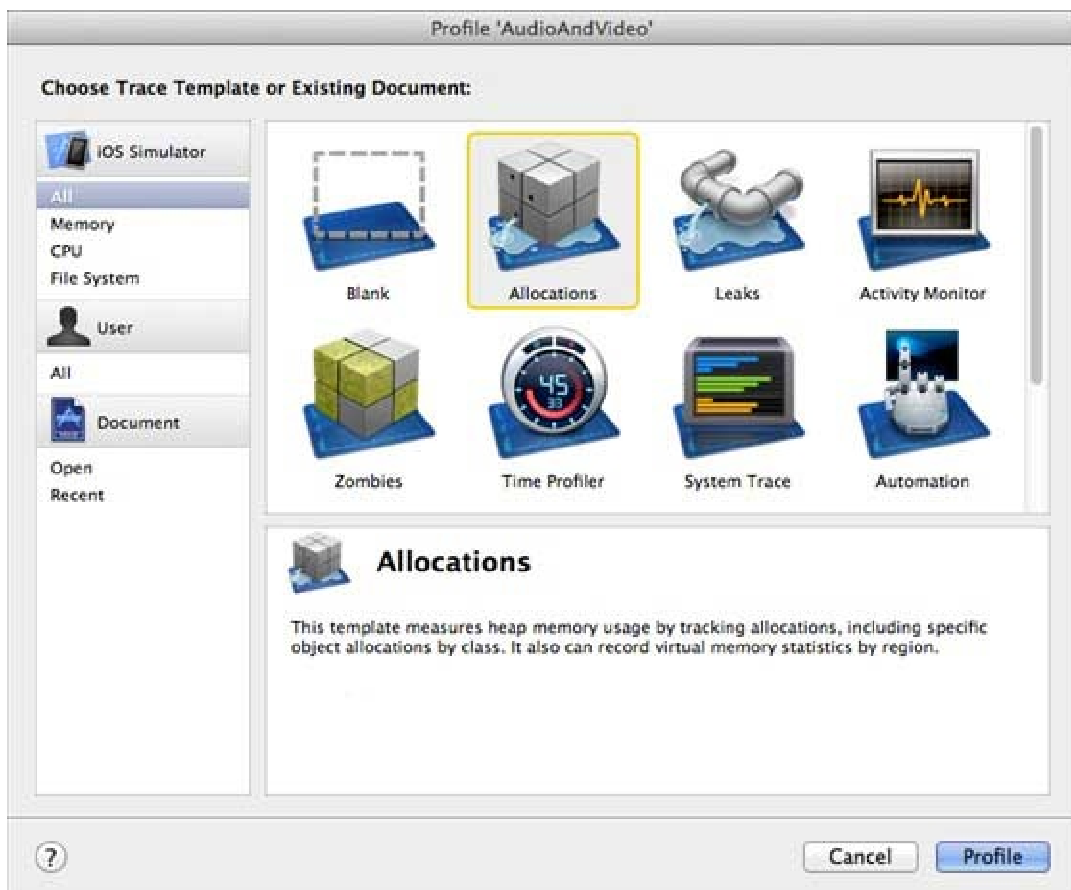
分析内存分配的步骤

1. 打开一个现有的应用程序。

2. 选择产品，配置文件如下所示

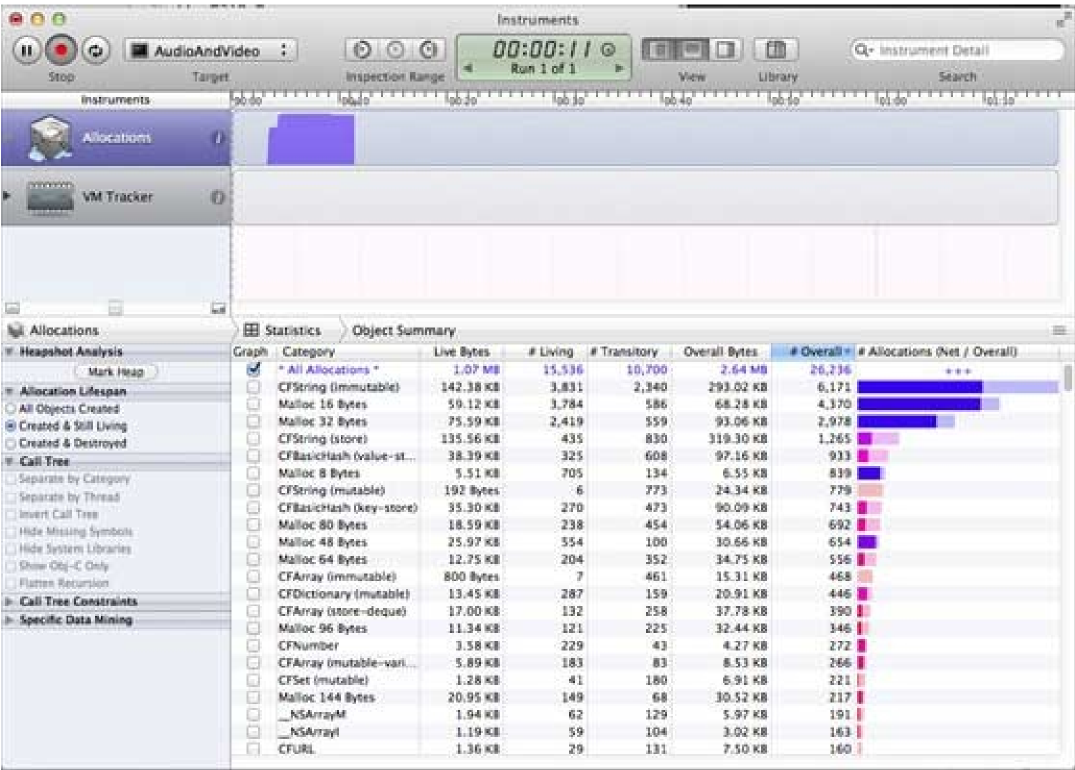


3.在以下界面中选择 Allocations 和 Profile。

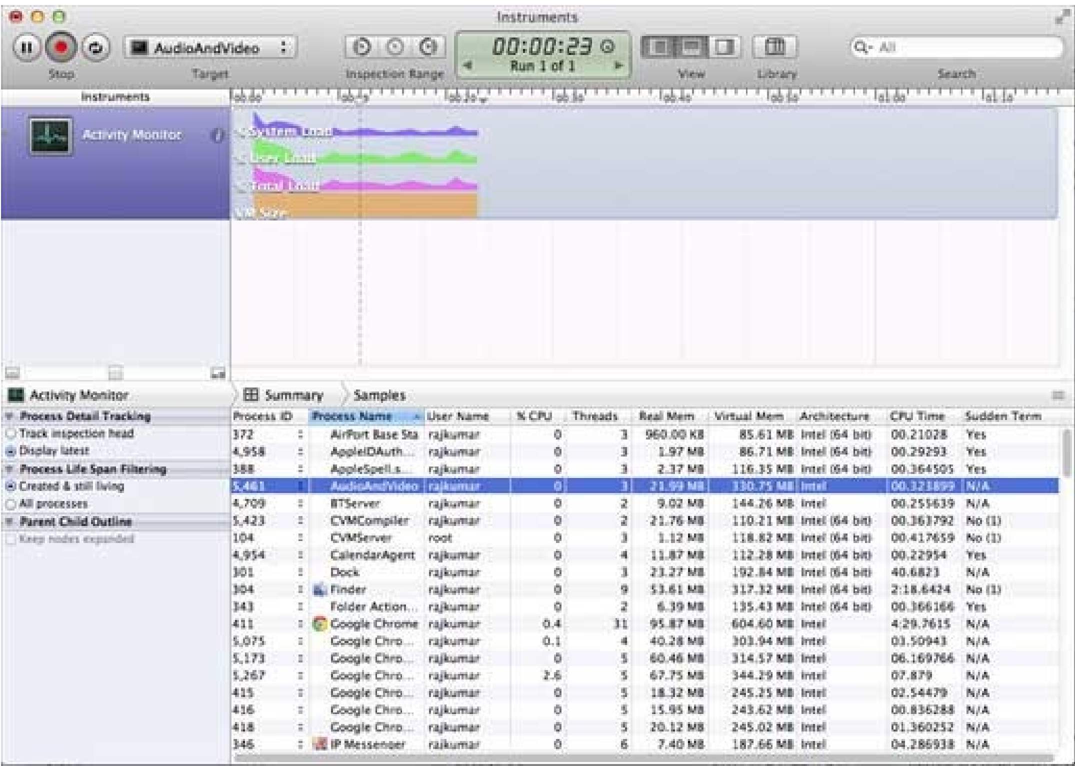


4. 我们可以看到不同对象的内存使用情况

5. 你可以切换视图控制器查看内存是否释放。



6.同样我们可以使用 Activity Monitor 来查看内存在应用程序中的分配的情况。



7. 这些工具可以帮助我们了解内存的使用情况及在什么地方可能发生泄漏。

IOS应用程序调试

简介

当我们做应用程序的时候，可能会犯各种错误，这可能会导致各种不同的错误。因此，为了修复这些错误或缺陷，我们需要来调试应用程序。

选择一个调试器

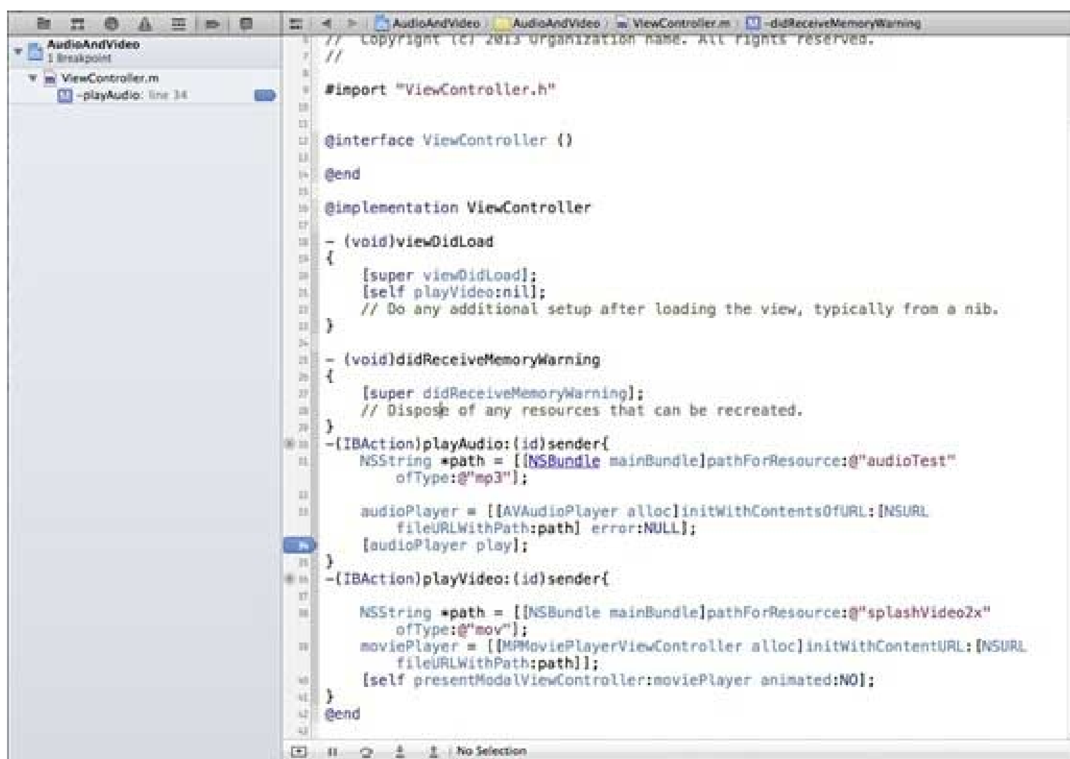
Xcode中调试器即 GDB 和 LLDB 调试器，GDB 是默认的。 LLDB是一个调试器是LLVM开源的编译器项目的一部分。您可以更改调试，编辑活动计划选项。

如何查找编码错误？

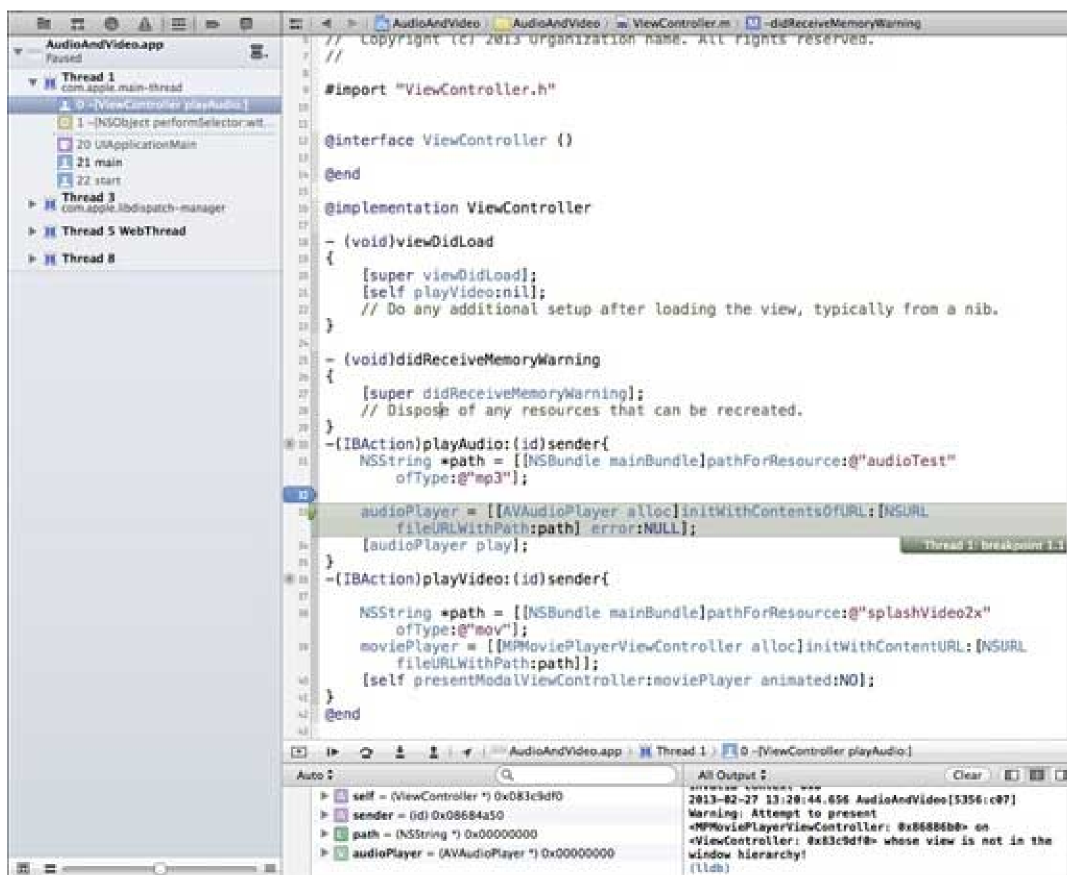
我们只需要建立我们的应用程序，代码被编译器编译，所有的消息，错误和警告将显示以及错误的原因，我们可以纠正他们。可以点击 product，然后点击"分析"，将在应用程序中可能发生的问题。

设置断点

断点帮助我们了解我们的应用程序对象，帮助我们找出许多缺陷，包括逻辑问题的不同状态。我们只需要点击创建一个断点的行号。我们可以通过点击并拖动它删除断点。如下所示



当我们运行应用程序并选择playVideo，按钮的应用程序将被暂停，我们来分析一下我们的应用程序的状态。当断点被触发时，我们将得到一个输出，如下图所示



可以轻松地确定哪个线程触发断点。在底部可以看到对象，如self，sender等，这些持有相应的对象的值，我们可以展开一些这些对象，看看他们每个的状态是什么。

要继续应用程序，我们在调试区选择继续按钮（最左边的按钮），如下图所示。其他选项包括步骤和单步跳过

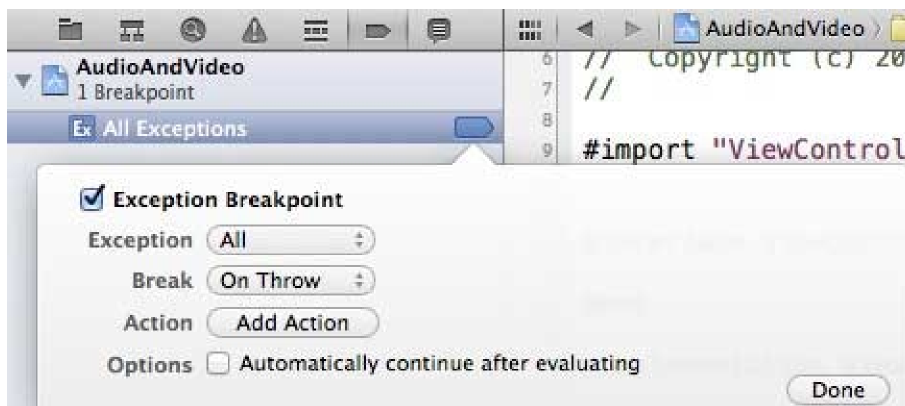


异常断点

我们也有异常断点，触发应用程序停止发生异常的位置。通过选择调试导航后选择"+"按钮，我们可以创建异常断点。将得到下面的窗口



然后，我们需要选择" Exception Breakpoint (添加异常)"断点，它会显示下面的窗口



下一步是什么？

你可以在 Xcode 4 用户指南 知道更多关于调试和其他Xcode功能的知识。

w3school jQuery Mobile 教程

来源：jQuery Mobile 教程

整理：飞龙

jQuery Mobile 简介

jQuery Mobile 是一个用于创建移动端web应用的的前端框架。

学习本教程前你需要先了解

在开始学习 jQuery Mobile 前, 你应该了解一下基础知识：

- HTML
- CSS
- jQuery

如果你想学习这些知识，你可以访问本站的首页。

什么是 jQuery Mobile?

jQuery Mobile 是针对触屏智能手机与平板电脑的网页开发框架。

jQuery Mobile 工作与所有主流的智能手机和平板电脑上：



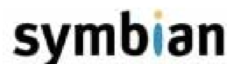














jQuery Mobile 构建于 jQuery 以及 jQuery UI类库之上，如果您了解 jQuery，您可以很容易的学习 jQuery Mobile。

jQuery Mobile 使用了极少的 HTML5、CSS3、JavaScript 和 AJAX 脚本代码来完成页面的布局渲染。

为什么使用 jQuery Mobile?

通过使用jQuery Mobile 可以 "写更少的代码，做更多的事情"：它可以通过一个灵活及简单的方式来布局网页，且兼容所有移动设备。

>  不同设备使用了不同开发语言，jQuery Mobile可以很好的兼容不容的设备或操作系统：

- Android 和 Blackberry（黑莓） 使用JAVA语言。
- iOS 使用 Objective C 语言
- Windows Phone 使用 C# 和 .net, 等。

> **jQuery Mobile** 解决了不同设备兼容的问题，因为它只使用**HTML**，**CSS**和**JavaScript**，这是所有移动网络浏览器的标准！

最好的阅读体验

尽管jQuery Mobile兼容所有的移动设备，但是并不能完全兼容PC机（由于有限的CSS3支持）。

为了更好的阅读本教程，建议您使用 **Google Chrome** 浏览器。

jQuery Mobile 安装

在你的网页中添加 jQuery Mobile

你可以通过以下几种方式将jQuery Mobile添加到你的网页中：

- 从 CDN 中加载 jQuery Mobile (推荐)
- 从jQuerymobile.com 下载 jQuery Mobile库

从 CDN 中加载 jQuery Mobile



CDN的全称是Content Delivery Network，即内容分发网络。其基本思路是尽可能避开互联网上有可能影响数据传输速度和稳定性的瓶颈和环节，使内容传输的更快、更稳定。.

使用 jQuery 内核, 你不需要在电脑上安装任何东西; 你仅仅需要在你的网页中加载以下层叠样式 (.css) 和 JavaScript 库 (.js) 就能够使用 jQuery Mobile:

jQuery Mobile CDN:

```
<head>
<link rel="stylesheet" href="http://code.jquery.com/mobile/1.3.2/jq
<script src="http://code.jquery.com/jquery-1.8.3.min.js"></script>
<script src="http://code.jquery.com/mobile/1.3.2/jquery.mobile-1.3.
</head>
```

下载 jQuery Mobile

如果你想将 jQuery Mobile 放于你的主机中,你可以从 jQuerymobile.com下载该文件。

```
<head>
<link rel=stylesheet href=jquery.mobile-1.3.2.css>
<script src=jquery.js></script>
<script src=jquery.mobile-1.3.2.js></script>
</head>
```

提示： 将下载的文件放置于与网页同一目录下。

 你是否想知道为什么在 **<script>** 标签中 没有插入 **type="text/javascript" ?**

在 HTML5 已经不需要该属性。 JavaScript 在所有现代浏览器中是 HTML5 的默认脚本语言！

jQuery Mobile 页面

开始学习 jQuery Mobile

>  尽管jQuery Mobile兼容所有的移动设备，但是并不能完全兼容PC机（由于有限的CSS3支持）。
>
> 为了更好的阅读本教程，建议您使用 Google Chrome 浏览器。

实例

```
<body>
<div data-role="page">

  <div data-role="header">
    <h1>欢迎来到我的主页</h1>
  </div>

  <div data-role="content">
    <p>我现在是一个移动端开发者!!</p>
  </div>

  <div data-role="footer">
    <h1>底部文本</h1>
  </div>

</div>
</body>
```

实例解析：

- data-role="page" 是在浏览器中显示的页面。
- data-role="header" 是在页面顶部创建的工具条 (通常用于标题或者搜索按钮)
- data-role="content" 定义了页面的内容，比如文本， 图片，表单，按钮等。
- data-role="footer" 用于创建页面底部工具条。
- 在这些容器中你可以添加任何 HTML 元素 - 段落, 图片, 标题, 列表等。

>  jQuery Mobile 依赖 HTML5 data-* 属性来支持各种 UI 元素、过渡和页面结构。不支持它们的浏览器将以静默方式弃用它们。

在页面中添加 jQuery Mobile

使用 jQuery Mobile, 你可以再单个 HTML 文件中创建多个不同的页面。

你可以使用不同的href属性来区分使用同一个唯一id的页面：

实例

```
<div data-role="page" id="pageone">
  <div data-role="content">
    <a href="#pagetwo">Go to Page Two</a>
  </div>
</div>

<div data-role="page" id="pagetwo">
  <div data-role="content">
    <a href="#pageone">Go to Page One</a>
  </div>
</div>
```

注意： 当web应用有大量的内容（文本，图片，脚本等）将会严重影响加载时间。如果你不想使用内页链接可以使用外部文件：

```
<a href="externalfile.html">访问外部文件</a>
```

页面作为对话框使用

对话框是用于显示页面信息显示或者表单信息的输入。

在链接中添加data-rel="dialog"让用户点击链接时弹出对话框:

实例

```
<div data-role="page" id="pageone">
  <div data-role="content">
    <a href="#pagetwo" data-rel="dialog">Go to Page Two</a>
  </div>
</div>

<div data-role="page" id="pagetwo">
  <div data-role="content">
    <a href="#pageone">Go to Page One</a>
  </div>
</div>
```

jQuery Mobile 页面切换

jQuery Mobile 包含 CSS3 效果让您选择页面打开的方式。

jQuery Mobile 页面切换效果

jQuery Mobile 提供了各种页面切换到下一个页面的效果。

注意：为了实现页面切换效果，浏览器必须支持 CSS3 3D 切换：

- Internet Explorer 10 支持 3D 切换（早期版本不支持）
- Opera 不支持 3D 切换

页面切换效果可被应用于任何使用 data-transition 属性的链接或表单提交：

```
<a href="#anylink" data-transition="slide">切换到第二个页面</a>
```

下面的表格显示了通过使用 data-transition 属性后可用的页面切换：

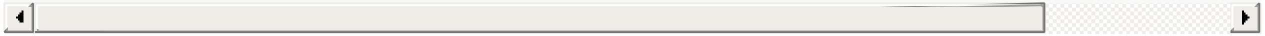
页面切换	描述	尝试一下
fade	默认。淡入到下一页	
flip	从后向前翻转到下一页	
flow	抛出当前页，进入下一页	
pop	像弹出窗口一样进入下一页	
slide	从右到左滑动到下一页	
slidefade	从右到左滑动并淡入到下一页	
slideup	从下到上滑动到下一页	
slidedown	从上到下滑动到下一页	
turn	翻到下一页	
none	没有切换效果	

 在 jQuery Mobile 的所有链接上，默认使用淡入淡出的效果（如果浏览器支持）。

提示：上面的所有效果支持后退行为。例如，如果您想要页面从左向右滑动，而不是从右向左滑动，请使用带有 "reverse" 值的 data-direction 属性。在后退按钮上这是默认的。

实例

```
<a href="#pagetwo" data-transition="slide" data-direction="reverse"
```



jQuery Mobile 按钮

Mobile 应用程序是建立在您想要显示的简单的点击事物上。



在 jQuery Mobile 中创建按钮

在 jQuery Mobile 中，按钮可通过三种方式创建：

- 使用 <button> 元素
- 使用 <input> 元素
- 使用带有 data-role="button" 的 <a> 元素

<button>

```
<button>按钮</button>
```

<input>

```
<input type="button" value="按钮">
```

<a>

```
<a href="#" data-role="button">按钮</a>
```

 在 jQuery Mobile 中，按钮会自动样式化，让它们在移动设备上更具吸引力和可用性。我们推荐您使用带有 **data-role="button"** 的 **<a>** 元素在页面间进行链接，使用 **<input>** 或 **<button>** 元素进行表单提交。

导航按钮

如需通过按钮在页面间进行链接，请使用带有 data-role="button" 属性的 <a> 元素：

实例

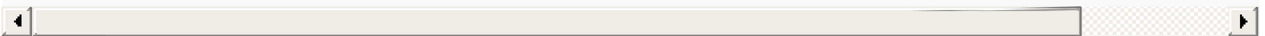
```
<a href="#pagetwo" data-role="button">访问第二个页面</a>
```

内联按钮

默认情况下，按钮占满整个屏幕宽度。如果你想要一个仅是与内容一样宽的按钮，或者如果您想要并排显示两个或多个按钮，请添加 data-inline="true"：

实例

```
<a href="#pagetwo" data-role="button" data-inline="true">访问第二个
```



组合按钮

jQuery Mobile 提供了一个简单的方法来将按钮组合在一起。

请把 data-role="controlgroup" 属性和 data-type="horizontal|vertical" 一起使用来规定是否水平或垂直组合按钮：

实例

```
<div data-role="controlgroup" data-type="horizontal">
<a href="#anylink" data-role="button">按钮 1</a>
<a href="#anylink" data-role="button">按钮 2</a>
<a href="#anylink" data-role="button">按钮 3</a>
</div>
```

 默认情况下，组合按钮是垂直组合，它们之间没有外边距和空间。并且只有第一个和最后一个按钮是圆角，以便它们组合在一起的时候创建一个漂亮的外观。

后退按钮

如需创建后退按钮，请使用 data-rel="back" 属性（这会忽略锚的 href 值）：

实例

```
<a href="#" data-role="button" data-rel="back">返回</a>
```

更多用于按钮的 data-* 属性

属性	值	描述	实例
data-corners	true | false	规定按钮是否圆角	
data-mini	true | false	规定按钮是否更小	
data-shadow	true | false	规定按钮是否有阴影	

如需查看所有 jQuery Mobile data-* 属性的完整参考手册，请访问我们的 jQuery Mobile Data 属性参考手册。

下一章演示如何附加图标到您的按钮上。

jQuery Mobile 按钮图标

jQuery Mobile 提供了一套让按钮看起来更称心如意的图标。



添加图标到 jQuery Mobile 按钮

如需添加图标到您的按钮，请使用 data-icon 属性：

```
<a href="#anylink" data-role="button" data-icon="search">Search</a>
```

提示：您也可以在 <button> 或 <input> 元素上使用 data-icon 属性。

下面我们列出一些 jQuery Mobile 提供的可用图标：

属性值	描述	图标	实例
data-icon="arrow-l"	左箭头		
data-icon="arrow-r"	右箭头		
data-icon="delete"	删除		
data-icon="info"	信息		
data-icon="home"	首页		
data-icon="back"	后退		
data-icon="search"	搜索		
data-icon="grid"	网格		

如需查看所有 jQuery Mobile 按钮图标的完整参考手册，请访问我们的 jQuery Mobile 图标参考手册。

定位图标

您也可以规定图标定位在按钮的什么部位：顶部（top）、右侧（right）、底部（bottom）、左侧（left）。

请使用 data-iconpos 属性来指定位置：

图标的位置：

```
<a href="#link" data-role="button" data-icon="search" data-iconpos=
<a href="#link" data-role="button" data-icon="search" data-iconpos=
<a href="#link" data-role="button" data-icon="search" data-iconpos=
<a href="#link" data-role="button" data-icon="search" data-iconpos=
```

 默认情况下，所有的按钮图标被放置到左侧。

只显示图标

如果只想显示图标，请设置 data-iconpos 为 "notext"：

后退：

```
<a href="#link" data-role="button" data-icon="search" data-iconpos=
```

jQuery Mobile 工具栏

jQuery Mobile 工具栏

工具栏元素通常位于头部和尾部内 - 让导航易于访问：

这是一个简单的弹窗！
阅读教程了解如何使用弹窗！

头部栏

头部栏一般包含页面标题/logo 或一两个按钮（通常是首页、选项或搜索）。

您可以添加按钮到头部的左侧或右侧。

下面的代码，将添加一个按钮到头部标题文本的左侧，添加一个按钮到头部标题文本的右侧：

实例

```
<div data-role="header">
<a href="#" data-role="button">首页</a>
<h1>欢迎来到我的主页</h1>
<a href="#" data-role="button">搜索</a>
</div>
```

下面的代码，将添加一个按钮到头部标题文本的左侧：

```
<div data-role="header">
<a href="#" data-role="button">首页</a>
<h1>欢迎来到我的主页</h1>
</div>
```

但是，如果您把按钮链接放置在 <h1> 元素之后，将无法显示右侧的文本。要添加一个按钮到头部标题的右侧，请指定 class 为 "ui-btn-right"：

实例

```
<div data-role="header">
<h1>欢迎来到我的主页</h1>
<a href="#" data-role="button" class="ui-btn-right">搜索</a>
</div>
```

 头部可以包含一个或两个按钮，而尾部没有限制。

尾部栏

尾部栏比头部栏更灵活 - 在整个页面中它们更具功能性和可变性，因此可以包含尽可能多的按钮：

实例

```
<div data-role="footer">
<a href="#" data-role="button">在 Facebook上关注我</a>
<a href="#" data-role="button">在Twitter上关注我</a>
<a href="#" data-role="button">在Instagram上关注我</a>
</div>
```

注意：尾部的样式与头部不同（没有内边距和空间，且按钮不居中）。为了解决这个问题，请把 "ui-btn" 放置在尾部的 class 上：

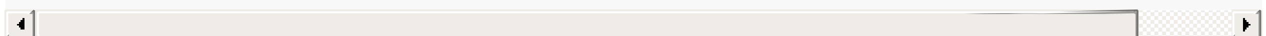
实例

```
<div data-role="footer" class="ui-btn">
```

您还可以将尾部中的按钮进行水平或垂直组合：

实例

```
<div data-role="footer" class="ui-btn">
<div data-role="controlgroup" data-type="horizontal">
<a href="#" data-role="button" data-icon="plus">在Facebook上关注我</
<a href="#" data-role="button" data-icon="plus">在Twitter上关注我</a
<a href="#" data-role="button" data-icon="plus">在Instagram上关注我<
</div>
</div>
```



定位头部栏和尾部栏

头部和尾部可以通过三种方式进行定位：

- Inline - 默认。头部栏和尾部栏与页面内容内联。
- Fixed - 头部栏和尾部栏固定在页面的顶部和底部。
- Fullscreen - 与 Fixed 定位模式基本相同，头部栏和尾部栏固定在页面的顶部和底部。但是当他工具栏滚动出屏幕之外时，不会自动重新显示，除非点击屏幕，这对于图片或视频类有提升代入感的应用是非常有用的。注意这种模式下工具栏会遮住页面内容，所以最好用在比较特殊的场合下。

使用 data-position 属性来定位头部栏和尾部栏：

Inline 定位（默认）

```
<div data-role="header" data-position="inline"></div>
<div data-role="footer" data-position="inline"></div>
```

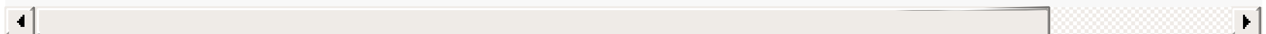
Fixed 定位

```
<div data-role="header" data-position="fixed"></div>
<div data-role="footer" data-position="fixed"></div>
```

要启用全屏定位，请使用 data-position="fixed"，并添加 data-fullscreen 属性到元素：

Fullscreen 定位

```
<div data-role="header" data-position="fixed" data-fullscreen="true
<div data-role="footer" data-position="fixed" data-fullscreen="true
```



提示：全屏定位适用于照片、图像和视频。

提示：固定定位和全屏定位中，通过点击屏幕将隐藏和显示头部栏和尾部栏。

jQuery Mobile 导航栏

jQuery Mobile 导航栏

导航栏是由一组水平排列的链接组成，通常包含在头部或尾部内。

默认情况下，导航栏中的链接将自动变成按钮（不需要 data-role="button"）。

使用 data-role="navbar" 属性来定义导航栏：

实例

```
<div data-role="header">
<div data-role="navbar">
<ul>
<li><a href="#anylink">首页</a></li>
<li><a href="#anylink">页面二</a></li>
<li><a href="#anylink">搜索</a></li>
</ul>
</div>
</div>
```

>  默认情况下，按钮的宽度与它的内容一样。使用一个无序列表来平均地划分按钮的宽度：1 个按钮占 100% 宽度，2 个按钮则各占 50% 的宽度，3 个按钮则每个占 33,3% 的宽度，依此类推。然而，如果您在导航栏中指定了超过 5 个按钮，将会拆成多行（查看"更多实例"）。

激活按钮

当导航栏中的某个链接被点击，它将获得被选中（按下）的外观。

如果想在不点击链接时获得这种外观，请使用 class="ui-btn-active"：

实例

```
<li><a href="#anylink" class="ui-btn-active">首页</a></li>
```

对于多个页面，您可能想要每个按钮的选中外观代表当前用户所在的页面。要做到这一点，请添加 "ui-state-persist" 和 "ui-btn-active" 到链接的 class：

实例

```
<li><a href="#anylink" class="ui-btn-active ui-state-persist">首页<
```

更多实例

内容中的导航栏 如何在 data-role="content" 内添加导航栏。

尾部中的导航栏 如何在尾部内添加导航栏。

导航栏中的定位图标 如何在尾部内的导航栏中定位图标。

超过 5 个按钮 导航栏中 10 个按钮的演示。

jQuery Mobile 可折叠块

可折叠内容块

可折叠块允许您隐藏或显示内容 - 对于存储部分信息很有用。

如需创建一个可折叠的内容块，需要为容器添加 data-role="collapsible" 属性。在容器（div）内，添加一个标题元素（H1-H6），后跟您想要进行扩展的 HTML 标记：

实例

```
<div data-role="collapsible">
<h1>点击我 - 我可以折叠!</h1>
<p>我是可折叠的内容。</p>
</div>
```

默认情况下，内容是被折叠起来的。如需在页面加载时展开内容，请使用 data-collapsed="false"：

实例

```
<div data-role="collapsible" data-collapsed="false">
<h1>点击我 - 我可以折叠!</h1>
<p>I'm 现在我默认是展开的。</p>
</div>
```

嵌套可折叠块

可折叠的内容块是可以彼此嵌套的：

实例

```
<div data-role="collapsible">
<h1>点击我 - 我可以折叠!</h1>
<p>我是被展开的内容。</p>
<div data-role="collapsible">
<h1>点击我 - 我是嵌套的可折叠块！</h1>
<p>我是嵌套的可折叠块中被展开的内容。</p>
</div>
</div>
```

 可折叠的内容块可以根据您的需要进行多次嵌套。

可折叠集合

可折叠集合是将可折叠块组合在一起（就像手风琴一样）。当一个新的块被展开时，所有其他的块都会被折叠起来。

创建若干个可折叠的内容块，然后把可折叠内容块用带有 data-role="collapsible-set" 的新容器包围起来：

实例

```
<div data-role="collapsible-set">
<div data-role="collapsible">
<h1>点击我 - 我可以折叠！</h1>
<p>我是被展开的内容。</p>
</div>
<div data-role="collapsible">
<h1>点击我 - 我可以折叠!</h1>
<p>我是被展开的内容。</p>
</div>
</div>
```

更多实例

通过 data-inset 属性取消圆角 如何取消可折叠块上的圆角。

通过 data-mini 属性迷你化可折叠块 如何让可折叠块更小。

通过 data-collapsed-icon 和 data-expanded-icon 改变图标 如何改变可折叠块的图标（默认是 + 和 -）。

jQuery Mobile 网格

jQuery Mobile 布局网格

jQuery Mobile 提供了一套基于 CSS 的分列布局。然而，在移动设备上，由于考虑手机的屏幕宽度狭窄，一般不建议使用分栏分列布局。

但有时您想要将较小的元素（如按钮或导航标签）并排地排列在一起，就像是在一个表格中一样。这种情况下，推荐使用分列布局。

网格中的列是等宽的（合计是 100%），没有边框、背景、margin 或 padding。

这里有四种布局网格可供使用：

网格类	列	列宽	对应	实例
ui-grid-a	2	50% / 50%	ui-block-a|b	
ui-grid-b	3	33% / 33% / 33%	ui-block-a|b|c	
ui-grid-c	4	25% / 25% / 25% / 25%	ui-block-a|b|c|d	
ui-grid-d	5	20% / 20% / 20% / 20% / 20%	ui-block-a|b|c|d|e	

 在列容器内，子元素拥有的 class 为 ui-block-a|b|c|d|e 取决于列数。列会浮动并排。

实例 **1**:class 为 ui-grid-a（两列布局），您必须指定 ui-block-a 和 ui-block-b 的两个子元素。

实例 **2**:class 为 ui-grid-b（三列布局），则必须添加 ui-block-a、ui-block-b 和 ui-block-c 的三个子元素。

自定义网格

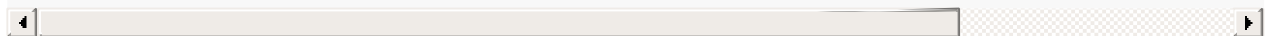
通过使用CSS，您可以自定义列块：

实例

```
<style>
.ui-block-a,
.ui-block-b,
.ui-block-c
{
background-color: lightgray;
border: 1px solid black;
height: 100px;
font-weight: bold;
text-align: center;
padding: 30px;
}
</style>
```

您也可以通过使用内嵌样式来自定义块：

```
<div class="ui-block-a" style="border: 1px solid black;"><span>Text
```



多行

在列中也可以有多个行。

注意：ui-block-a-class 总是创建一个新行：

实例

```
<div class="ui-grid-b">
<div class="ui-block-a"><span>Some Text</span></div>
<div class="ui-block-b"><span>Some Text</span></div>
<div class="ui-block-c"><span>Some Text</span></div>
<div class="ui-block-a"><span>Some Text</span></div>
<div class="ui-block-b"><span>Some Text</span></div>
<div class="ui-block-a"><span>Some Text</span></div>
</div>
```

jQuery Mobile 列表视图



jQuery Mobile 列表视图

jQuery Mobile 中的列表视图是标准的HTML 列表; 有序(<ol>) 和 无序(<ul>).

列表视图是jQuery Mobile中功能强大的一个特性。它会使标准的无序或有序列表应用更广泛。应用方法就是在ul或ol标签中添加data-role="listview"属性。在每个项目(<li>)中添加链接，用户可以点击它：

实例

```
<ol data-role="listview">
  <li><a href="#">列表项m</a></li>
</ol>

<ul data-role="listview">
  <li><a href="#">列表项</a></li>
</ul>
```

列表样式的圆角和边缘，使用 data-inset="true" 属性设置:

实例

```
<ul data-role="listview" data-inset="true">
```

>  默认情况下，列表项的链接会自动变成一个按钮 (不需要 data-role="button")。 |

列表分隔

列表项也可以转化为列表分割项，用来组织列表，使列表项成组。

指定列表分割，给列表项<li>元素添加 data-role="list-divider" 属性即可：

实例

```
<ul data-role="listview">
 <li data-role="list-divider">欧洲</li>
  <li><a href="#">法国</a></li>
  <li><a href="#">德国</a></li>
</ul>
```

如果你有一个字母顺序排列的列表，（例如一个电话簿）通过 <ol> 或者 元素的 data-autodividers="true" 属性设置可以配置为自动生成的项目的分隔:

实例

```
<ul data-role="listview" data-autodividers="true">
  <li><a href="#">Adele</a></li>
  <li><a href="#">Agnes</a></li>
  <li><a href="#">Billy</a></li>
  <li><a href="#">Calvin</a></li>
  ...
</ul>
```

>  data-autodividers="true" 可以配置为自动生成的项目的分隔。默认情况下，创建的分隔文本是列表项文本的第一个大写字母。

搜索过滤

jquery Mobile提供一个非常简单的方法，实现客户端搜索功能，筛选列表的选项。只需添加 data-filter="true" 属性即可:

实例

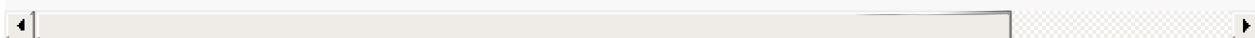
```
<ul data-role="listview" data-filter="true"></ul>
```

默认情况下，搜索输入框默认的字符为 "Filter items..."。

通过设置mobileinit事件的绑定程序或者给 $.mobile.listview.prototype.options.filterPlaceholder 选项设置一个字符串，或者给列表设置 data-filter-placeholder 属性，可以设置搜索输入框的默认字符:

实例

```
<ul data-role="listview" data-filter="true" data-filter-placeholder
```



更多实例

只读列表 如何创建一个不带链接的列表 (不会是个按钮且不可点击)。

jQuery Mobile 列表内容

浏览器

Google Chrome

Google Chrome 是免费的开源 web 浏览器。发布于 2008 年。

Mozilla Firefox

Firefox 是来自 Mozilla 的 web 浏览器。发布于 2004 年。

jQuery Mobile 列表缩略图

大于 16x16px 的图像，请在链接中添加 <img> 元素。

jQuery Mobile 将自动缩放图像到 80x80px:

实例

```
<ul data-role="listview">
  <li><a href="#"><img src="chrome.png"></a></li>
</ul>
```

使用标准的HTML添加列表信息：

实例

```
<ul data-role="listview">
  <li>
    <a href="#">
    <img src="chrome.png">
    <h2>Google Chrome</h2>
    <p>Google Chrome 免费的开源 web 浏览器。发布于 2008 年。</p>
    </a>
  </li>
</ul>
```

jQuery Mobile 列表图标

在列表 <img> 元素使用 class="ui-li-icon" 添加 16x16px 图标：

实例

```
<li><a href="#"><img src="us.png" alt="USA" class="ui-li-icon">USA<
```

分割按钮

在JQuery Mobile的列表中，有时需要对选项内容做两个不同的操作，这时，需要对选项中的链接按钮进行分割。实现分割的方法是在<li>元素中再增加一个<a>元素，便可以在页面实现分割效果。

jQuery Mobile 会自动设置第二个链接为蓝色箭头的图标，图标的链接文字（如果有的话）将在用户将鼠标悬停在 图标时显示:

实例

```
<ul data-role="listview">
  <li>
    <a href="#"><img src="chrome.png"></a>
    <a href="#">Some Text</a>
  </li>
</ul>
```

添加一些页面和对话框使链接功能更加丰富：

实例

```
<ul data-role="listview">
  <li>
    <a href="#"><img src="chrome.png"></a>
    <a href="#download" data-rel="dialog">下载浏览器</a>
  </li>
</ul>
```

气泡数字

气泡数字是用来显示列表项相关的数字，如在一个邮箱的邮件：

如需添加气泡数字，请使用行内元素，比如 <span>，设置 class "ui-li-count" 属性并添加数字：

实例

```
<ul data-role="listview">
  <li><a href="#">收件箱<span class="ui-li-count">25</span></a></li>
  <li><a href="#">发件箱<span class="ui-li-count">432</span></a></li>
  <li><a href="#">垃圾箱<span class="ui-li-count">7</span></a></li>
</ul>
```

注意：显示一个正确的气泡数字，必须修改编程方式。 这将在以后的章节解释。

更多实例

改变列表项的默认链接图标 如何设置列表项的默认链接图标(默认是右箭头).

可折叠的列表 如何创建显示/隐藏的列表。

更多内容格式 如何制作一个日历。

jQuery Mobile 表单

jQuery Mobile 会自动为 HTML 表单自动添加样式，让它们看起来更具吸引力，触摸起来更具友好性。

jQuery Mobile 表单

全名： 你的姓名是？

需要查找什么？ 搜索内容

今天的日期： 年 /月/日

选择喜爱的颜色： 红色

切换开关： Off

选择喜欢的电影：

蜘蛛侠

变形金刚

碟中谍

jQuery Mobile 表单结构

jQuery Mobile 使用 CSS 为 HTML 表单元素添加样式，让它们更具吸引力，更易于使用。

在 jQuery Mobile 中，您可以使用下列表单控件：

- 文本输入框
- 搜索输入框
- 单选按钮
- 复选框
- 选择菜单
- 滑动条
- 翻转拨动开关

当使用 jQuery Mobile 表单时，您应当知道：

- <form> 元素必须有一个 method 和一个 action 属性
- 每个表单元素必须有一个唯一的 "id" 属性。id 必须是整个站点所有页面上唯一的。这是因为 jQuery Mobile 的单页导航机制使得多个不同页面在同一时间被呈现
- 每个表单元素必须有一个标签。设置标签的 **for** 属性来匹配元素的 id

实例

```
<form method="post" action="demoform.html">
<label for="fname">姓名:</label>
<input type="text" name="fname" id="fname">
</form>
```

如需隐藏标签，请使用 class ui-hidden-accessible。这在您把元素的 placeholder 属性作为标签时经常用到：

实例

```
<form method="post" action="demoform.html">
<label for="fname" class="ui-hidden-accessible">姓名:</label>
<input type="text" name="fname" id="fname" placeholder="姓名...">
</form>
```

Field 容器

如需让标签和表单元素看起来更适应宽屏，请用带有 data-role="fieldcontain" 属性的 <div> 或 <fieldset> 元素包围 label/form 元素：

实例

```
<form method="post" action="demoform.html">
<div data-role="fieldcontain">
<label for="fname">姓:</label>
<input type="text" name="fname" id="fname">
<label for="lname">名:</label>
<input type="text" name="lname" id="lname">
</div>>
</form>
```

>  fieldcontain 属性基于页面的宽度为标签和表单控件添加样式。当页面的宽度大于 480px 时，它会自动把标签放置在与表单控件同一线上。当页面的宽度小于 480px 时，标签会被放置在表单元素的上面。

提示：为了防止 jQuery Mobile 为可点击元素自动添加样式，请使用 data-role="none" 属性：

实例

```
<label for="fname">姓名:</label>
<input type="text" name="fname" id="fname" data-role="none">
```

>  **jQuery Mobile 中的表单提交**
>
> jQuery Mobile 通过 AJAX 自动处理表单提交，并将试图集成服务器响应到应用程序的 DOM 中。

jQuery Mobile 表单输入元素

jQuery Mobile 文本输入框

输入字段是通过标准的 HTML 元素编码的，jQuery Mobile 将为它们添加样式使其看起来更具吸引力，在移动设备上更易使用。您也能使用新的 HTML5 的 <input> 类型：

实例

```
<form method="post" action="demo_form.php">
<div data-role="fieldcontain">
<label for="fullname">全名:</label>
<input type="text" name="fullname" id="fullname">

<label for="bday">生日:</label>
<input type="date" name="bday" id="bday">

<label for="email">E-mail:</label>
<input type="email" name="email" id="email" placeholder="你的电子邮箱
</div>
</form>
```

提示：请使用 placeholder 来指定一个简短的描述，用来描述输入字段的期望值：

```
<input placeholder="_sometext_">
```

文本域

对于多行文本输入可使用 <textarea> 。

注意：当您键入一些文本时，文本域会自动调整大小以适应新增加的行。

实例

```
<form method="post" action="demo_form.php">
<div data-role="fieldcontain">
<label for="info">附加信息:</label>
<textarea name="addinfo" id="info"></textarea>
</div>
</form>
```

搜索输入框

type="search" 类型的输入框是在 HTML5 中新增的，它是为输入搜索定义文本字段：

实例

```
<form method="post" action="demo_form.php">
<div data-role="fieldcontain">
<label for="search">搜索:</label>
<input type="search" name="search" id="search">
</div>
</form>
```

单选按钮

当用户在有限数量的选择中仅选取一个选项时，使用单选按钮。

为了创建一系列单选按钮，请添加带有 type="radio" 的 input 以及相应的 label。把单选按钮包围在 <fieldset> 元素内。您也可以添加一个 <legend> 元素来定义 <fieldset> 的标题。

提示：请使用 data-role="controlgroup" 来把按钮组合在一起：

实例

```
<form method="post" action="demo_form.php">
<fieldset data-role="controlgroup">
<legend>Choose your gender:</legend>
<label for="male">Male</label>
<input type="radio" name="gender" id="male" value="male">
<label for="female">Female</label>
<input type="radio" name="gender" id="female" value="female">
</fieldset>
</form>
```

复选框

当用户在有限数量的选择中选取一个或多个选项时，使用复选框：

实例

```
<form method="post" action="demo_form.php">
<fieldset data-role="controlgroup">
<legend>Choose as many favorite colors as you'd like:</legend>
<label for="red">Red</label>
<input type="checkbox" name="favcolor" id="red" value="red">
<label for="green">Green</label>
<input type="checkbox" name="favcolor" id="green" value="green">
<label for="blue">Blue</label>
<input type="checkbox" name="favcolor" id="blue" value="blue">
</fieldset>
</form>
```

更多实例

如需水平组合单选按钮或复选框，请使用 data-type="horizontal"：

实例

```
<fieldset data-role="controlgroup" data-type="horizontal">
```

您也可以用一个 field 容器包围 <fieldset>：

实例

```
<div data-role="fieldcontain">
<fieldset data-role="controlgroup">
<legend>请选择您的性别:</legend>
</fieldset>
</div>
```

如果您想要您的按钮中的一个预先选中，请使用 HTML 中 <input> 的 checked 属性：

实例

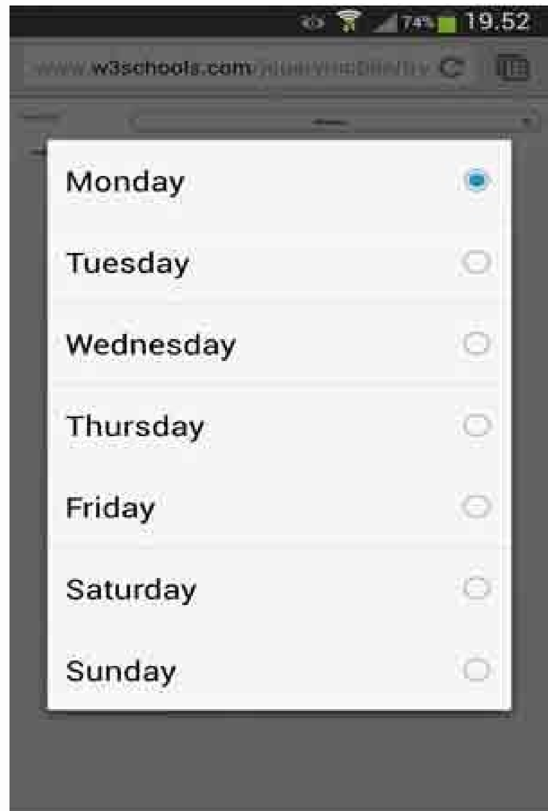
```
<input type="radio" checked>
<input type="checkbox" checked>
```

jQuery Mobile 表单选择菜单

jQuery Mobile 选择菜单

Iphone 上的选择菜单：Android/SGS4 设备上的选择菜单：





<select> 元素创建带有若干选项的下拉列表。

<select> 元素内的 <option> 元素定义了列表中的可用选项：

实例

```
<form method="post" action="demoform.html">
<fieldset data-role="fieldcontain">
<label for="day">Select Day</label>
<select name="day" id="day">
<option value="mon">Monday</option>
<option value="tue">Tuesday</option>
<option value="wed">Wednesday</option>
</select>
</fieldset>
</form>
```

提示：如果您有一个带有相关选项的很长的列表，请在 <select> 内使用 <optgroup> 元素：

实例

```
<select name="day" id="day">
<optgroup label="Weekdays">
<option value="mon">Monday</option>
<option value="tue">Tuesday</option>
<option value="wed">Wednesday</option>
</optgroup>
<optgroup label="Weekends">
<option value="sat">Saturday</option>
<option value="sun">Sunday</option>
</optgroup>
</select>
```

自定义选择菜单

本页顶部的图像，演示了移动平台上如何使用它们的方式展示一个选择菜单。

如果您想要让选择菜单在所有的移动设备上都显示相同，请使用 jQuery 自带的自定义选择菜单，data-native-menu="false" 属性：

实例

```
<select name="day" id="day" data-native-menu="false">
```

多个选择

如需在选择菜单中选择多个选项，请在 <select> 元素中使用 multiple 属性：

实例

```
<select name="day" id="day" multiple data-native-menu="false">
```

更多实例

使用 data-role="controlgroup" 如何组合一个或多个选择菜单。

使用 data-type="horizontal" 如何水平组合选择菜单。

预选中选项 如何预选中一个选项。

可折叠表单 如何创建可折叠表单。

jQuery Mobile 表单滑动条

jQuery Mobile 滑动条控件

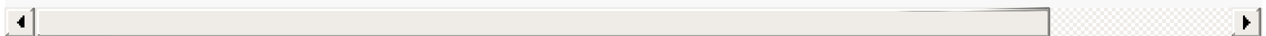
滑动条允许您从一个范围的数字中选择一个值：



如需创建滑动条，请使用 <input type="range">：

实例

```
<form method="post" action="demoform.html">
<div data-role="fieldcontain">
<label for="points">Points:</label>
<input type="range" name="points" id="points" value="50" min="0" ma
</div>
</form>
```



使用以下属性来规定限制：

- max - 规定允许的最大值
- min - 规定允许的最小值
- step - 规定合法的数字间隔
- value - 规定默认值

提示：如果您想要高亮突出显示滑动条的值，请添加 data-highlight="true"：

实例

```
<input type="range" data-hightlight="true">
```

拨动开关

波动开关通常用于 on/off 或 true/false 按钮：

如需创建一个开关，请把 <select> 元素与 data-role="slider" 一起使用，并添加两个 <option> 元素：

实例

```
<form method="post" action="demoform.html">
<div data-role="fieldcontain">
<label for="switch">Toggle Switch:</label>
<select name="switch" id="switch" data-role="slider">
<option value="on">On</option>
<option value="off">Off</option>
</select>
</div>
</form>
```

提示：请使用 "selected" 属性来设置选项中的一个为预选中状态（高亮突出显示状态）：

实例

```
<option value="off" selected>Off</option>
```

jQuery Mobile 主题

jQuery Mobile 主题

jQuery Mobile 提供了5种不同的主题样式, 从 "a" 到 "e" - 每一种主题的按钮，工具条，内容块等等颜色都不一致，每个主题的视觉效果也不一样。

通过设置元素的data-theme属性可以自定义应用的外观:

```
<div data-role="page" data-theme="a|b|c|d|e">
```

值	描述	实例
a	默认。黑色背景白色文字	
b	蓝色背景白色文字/ 黑色文字灰色背景	
c	黑色文字浅灰色背景	
d	黑色为主白色背景	
e	黑色文字橙色背景	

 你喜欢混合主题！

默认情况下， jQuery Mobile 使用 "a" 主题 (黑色) 来设置头部和底部， "c" 主题 (浅灰色) 设置页面内容。 但是，你可以自定义设置你喜欢的混合主题。

主题头部，内容和底部

实例

```
<div data-role="header" data-theme="b"></div>

<div data-role="content" data-theme="a"></div>

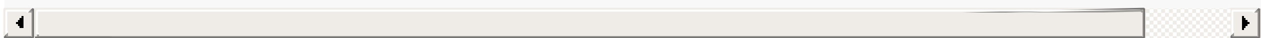
<div data-role="footer" data-theme="e"></div>
```

主题对话框

实例

```
<a href="#pagetwo" data-rel="dialog">Go To The Themed Dialog Page</

<div data-role="page" id="pagetwo" data-overlay-theme="e">
  <div data-role="header" data-theme="b"></div>
  <div data-role="content" data-theme="a"></div>
  <div data-role="footer" data-theme="c"></div>
</div>
```



主题按钮

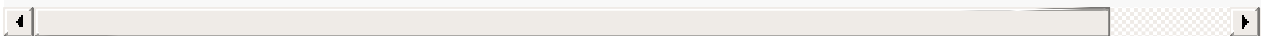
实例

```
<a href="#" data-role="button" data-theme="a">Button</a>
<a href="#" data-role="button" data-theme="b">Button</a>
<a href="#" data-role="button" data-theme="c">Button</a>
```

主题图标

实例

```
<a href="#" data-role="button" data-icon="plus" data-theme="e">Plus
```



头部和底部的主题按钮

实例

```
<div data-role="header">
  <a href="#" data-role="button" data-icon="home" data-theme="b">Ho
  <h1>Welcome To My Homepage</h1>
  <a href="#" data-role="button" data-icon="search" data-theme="e">
</div>

<div data-role="footer">
  <a href="#" data-role="button" data-theme="b" data-icon="plus">Bu
  <a href="#" data-role="button" data-theme="c" data-icon="plus">Bu
  <a href="#" data-role="button" data-theme="e" data-icon="plus">Bu
</div>
```

主题导航条

实例

```
<div data-role="footer" data-theme="e">
  <h1>Insert Footer Text Here</h1>
  <div data-role="navbar">
    <ul>
      <li><a href="#" data-icon="home" data-theme="b">Button 1</a><
      <li><a href="#" data-icon="arrow-r">Button 2</a></li>
      <li><a href="#" data-icon="arrow-r">Button 3</a></li>
      <li><a href="#" data-icon="search" data-theme="a" >Button 4</
    </ul>
  </div>
</div>
```

主题可折叠按钮和内容

实例

```
<div data-role="collapsible" data-theme="b" data-content-theme="e">
  <h1>Click me - I'm collapsible!</h1>
  <p>I'm the expanded content.</p>
</div>
```

主题列表

实例

```
<ul data-role="listview" data-theme="e">
  <li><a href="#">List Item</a></li>
  <li data-theme="a"><a href="#">List Item</a></li>
  <li data-theme="b"><a href="#">List Item</a></li>
  <li><a href="#">List Item</a></li>
</ul>
```

主题分割按钮

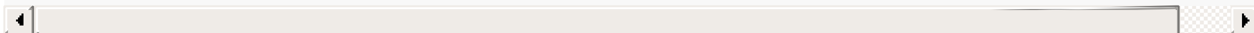
实例

```
<ul data-role="listview" data-split-theme="e">
```

主题可折叠列表

实例

```
<div data-role="collapsible" data-theme="b" data-content-theme="e">
  <ul data-role="listview">
    <li><a href="#">Agnes</a></li>
  </ul>
</div>
```



主题表单

实例

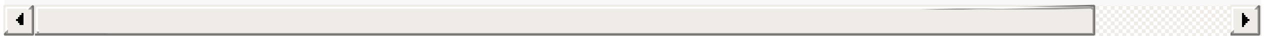
```
<label for="name">Full Name:</label>
<input type="text" name="text" id="name" data-theme="a">

<label for="colors">Choose Favorite Color:</label>
<select id="colors" name="colors" data-theme="b">
  <option value="red">Red</option>
  <option value="green">Green</option>
  <option value="blue">Blue</option>
</select>
```

主题可折叠表单

实例

```
<fieldset data-role="collapsible" data-theme="b" data-content-theme
<legend>Click me - I'm collapsible!</legend>
```



添加新主题

jQuery Mobile 可以在移动页面添加新主题。

通过修改 CSS 文件来添加或编辑新主题(如果你已经下载了 jQuery Mobile)。你只需要拷贝样式模块，然后重命令字母类名（f-z）,并在样式中添加你喜欢的颜色和字体。

你也可以在 HTML 文档中添加主题的新样式 - 工具条添加类 ： ui-bar-(a-z)， 文本内容添加类: ui-body-(a-z) for the content:

实例

```
<style>
.ui-bar-f
{
color:green;
background-color:yellow;
}
.ui-body-f
{
font-weight:bold;
color:purple;
}
</style>
```

jQuery Mobile 事件

事件 = 所有不同访问者访问页面的响应动作。

jQuery Mobile 事件

在jQuery Mobile你可以使用任何标准的 jQuery 事件 。

除此之外, jQuery Mobile 也提供了针对移动端浏览器的事件：

- 触摸事件 - 当用户触摸屏幕时触发
- 滑动事件 - 当用户上下滑动时触发
- 定位事件 - 当设备水平或垂直翻转时触发
- 页面事件 - 当页面显示，隐藏，创建，加载或未加载时触发

初始化 jQuery Mobile 事件

在学习jQuery时我们学到了用$(document).ready()来使你的jQuery代码脚本在DOM元素加载完成后才开始执行：

jQuery document ready 事件

```
<script>
$(document).ready(function(){

   _// jQuery methods go here...
_
});
</script>
```

但是，在 jQuery Mobile 中, 使用pageinit 事件来设置代码脚本在DOM元素加载完成后开始执行，所以要在任何新页面加载并创建是执行脚本，就需要绑定pageinit事件。

第二个参数 ("*#pageone*")为指定事件的页面id：

jQuery Mobile pageinit 事件

```
<script>
$(document).on("pageinit","#pageone",function(){

    _// jQuery events go here..._

});
</script>
```

>  注意： jQuery on() 方法用于绑定事件到选中的元素上。

下一章节我们将更详细介绍 jQuery Mobile 事件。

jQuery Mobile 触摸事件

触摸事件在用户触摸屏幕（页面）时触发。

点击或者滑动本区域

 触摸事件同样可应用与桌面电脑上：点击或者滑动鼠标！

jQuery Mobile 点击

点击事件在用户点击元素时触发。

如下实例：当点击 <p> 元素时，隐藏当前的 <p> 元素：

实例

```
$("p").on("tap",function(){
  $(this).hide();
});
```

jQuery Mobile 点击不放（长按）

点击不放（长按） 事件在点击并不放（大约一秒）后触发

实例

```
$("p").on("taphold",function(){
  $(this).hide();
});
```

jQuery Mobile 滑动

滑动事件是在用户一秒内水平拖拽大于30PX，或者纵向拖曳小于20px的事件发生时触发的事件：

实例

```
$("p").on("swipe",function(){
  $("span").text("Swipe detected!");
});
```

jQuery Mobile 向左滑动

向左滑动事件在用户向左拖动元素大于30px时触发：

实例

```
$("p").on("swipeleft",function(){
  alert("You swiped left!");
});
```

jQuery Mobile 向右滑动

向右滑动事件在用户向右拖动元素大于30px时触发：

实例

```
$("p").on("swiperight",function(){
  alert("You swiped right!");
});
```

jQuery Mobile 滚屏事件

jQuery Mobile 提供了两种滚屏事件：滚屏开始时触发和滚动结束时触发。

jQuery Mobile 滚屏开始（Scrollstart）

scrollstart 事件是在用户开始滚动页面时触发：

实例

```
$(document).on("scrollstart",function(){
alert("Started scrolling!");
});
```

>  注意：iOS 设备在滚屏时锁定 DOM 操作，这意味着当用户滚屏时不可能改变任何东西。然而，jQuery 团队正在为此寻找解决方案。

jQuery Mobile 滚屏结束（Scrollstop）

scrollstop 事件是在用户停止滚动页面时触发：

实例

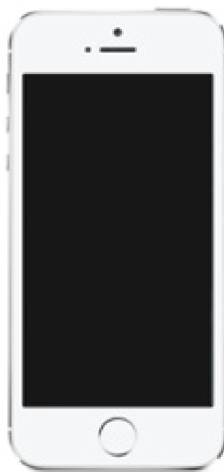
```
$(document).on("scrollstop",function(){
alert("Stopped scrolling!");
});
```

jQuery Mobile 方向改变事件

jQuery Mobile 方向改变（orientationchange）事件

当用户垂直或水平旋转移动设备时，触发方向改变（orientationchange）事件。

水平旋转 垂直旋转



如需使用方向改变（orientationchange）事件，请附加它到 window 对象：

```
$(window).on("orientationchange",function(){
alert("The orientation has changed!");
});
```

回调函数可有一个参数，event 对象，返回移动设备的方向："纵向"（设备保持在垂直位置）或"横向"（设备保持在水平位置）：

实例

```
$(window).on("orientationchange",function(event){
alert("Orientation is: " + event.orientation);
});
```

由于方向改变（orientationchange）事件绑定到 window 对象，我们可以使用 window.orientation 属性来设置不同的样式，以便区分纵向和横向的视图：

实例

```
$(window).on("orientationchange",function(){
if(window.orientation == 0) // Portrait
{
$("p").css({"background-color":"yellow","font-size":"300%"});
}
else // Landscape
{
$("p").css({"background-color":"pink","font-size":"200%"});
}
});
```

>  window.orientation 属性针对纵向视图返回 0，针对横向视图返回 90 或 -90。

jQuery Mobile Data 属性

jQuery Data 属性

jQuery Mobile 使用 HTML5 data-* 属性来为移动设备创建更具触摸友好性和吸引性的外观。

在下面的参考列表中，粗体显示的值为默认值。

按钮

带有 data-role="button" 的超链接。button 元素、工具栏中的链接以及 input 字段都会自动渲染成按钮样式，不需要添加 data-role="button"。

Data-属性	值	描述
data-corners	**true** | false	规定按钮是否圆角
data-icon	Icons 参考手册	规定按钮的图表。默认没有图标。
data-iconpos	**left** | right | top | bottom | notext	规定图标的位置
data-iconshadow	**true** | false	规定按钮图标是否有阴影
data-inline	true | **false**	规定按钮是否内联
data-mini	true | **false**	规定按钮是小尺寸还是常规尺寸
data-shadow	**true** | false	规定按钮是否有阴影
data-theme	*letter* (a-z)	规定按钮的主题颜色

 如需组合多个按钮，请使用带有 data-role="controlgroup" 属性和 data-type="horizontal|vertical" 的容器来规定是否水平或垂直组合按钮。

复选框

带有 type="checkbox" 的成双成对的 label 和 input。它们会被自动渲染程按钮样式，data-role 不是必需的。

Data-属性	值	描述
data-mini	true | **false**	规定复选框是小尺寸还是常规尺寸
data-role	none	防止 jQuery Mobile 把复选框渲染成按钮样式
data-theme	*letter* (a-z)	规定复选框的主题颜色

 如需组合多个复选框，请使用带有 data-role="controlgroup" 属性和 data-type="horizontal|vertical" 的容器来规定是否水平或垂直组合复选框。

可折叠块

在带有 data-role="collapsible" 的容器内部的后跟任意 HTML 标记的标题元素。

Data-属性	值	描述
data-collapsed	**true** | false	规定内容是折叠还是展开
data-collapsed-icon	Icons 参考手册	规定可折叠按钮的图标。默认是 "plus"
data-content-theme	*letter* (a-z)	规定可折叠内容的主题颜色。是否为可折叠内容添加圆角
data-expanded-icon	Icons 参考手册	规定当内容展开时可折叠按钮的图标。默认是 "minus"
data-iconpos	**left** | right | top | bottom	规定图标的位置
data-inset	**true** | false	规定可折叠按钮是否渲染成圆角且带外边距
data-mini	true | **false**	规定可折叠按钮是小尺寸还是常规尺寸
data-theme	*letter* (a-z)	规定可折叠按钮的主题颜色

可折叠集合

在带有 data-role="collapsible-set" 的容器内部的可折叠内容块。

Data-属性	值	描述
data-collapsed-icon	Icons 参考手册	规定可折叠按钮的图标。默认是 "plus"
data-content-theme	*letter* (a-z)	规定可折叠按钮的主题颜色
data-expanded-icon	Icons 参考手册	规定当内容展开时可折叠按钮的图标。默认是 "minus"
data-iconpos	**left** | right | top | bottom | notext	规定图标的位置
data-inset	**true** | false	规定可折叠块是否渲染成圆角且带外边距
data-mini	true | **false**	规定可折叠按钮是小尺寸还是常规尺寸
data-theme	*letter* (a-z)	规定可折叠集合的主题颜色

内容

带有 data-role="content" 的容器。

Data-属性	值	描述
data-theme	*letter* (a-z)	规定内容的主题颜色。默认是 "c"

控件组

带有 data-role="controlgroup" 的 <div> 或 <fieldset> 容器。 组合单个类型（基于链接的按钮、单选按钮、复选框、select 元素）的多个按钮样式的 input。对于组合表单复选框和单选按钮，推荐在带有 data-role="fieldcontain" 的 <div> 内使用 <fieldset> 容器来改进标签样式。

Data-属性	值	描述
data-mini	true | **false**	规定控件组是小尺寸还是常规尺寸
data-type	horizontal | **vertical**	规定控件组是水平显示还是垂直显示

对话框

带有 data-role="dialog" 的容器或带有 data-rel="dialog" 的链接。

Data-属性	值	描述
data-close-btn-text	*sometext*	规定对话框关闭按钮的文本
data-dom-cache	true | **false**	规定是否清除各个页面的 jQuery DOM 缓存（如果设置为 true，您必须自己管理 DOM 并在所有移动设备上进行测试）
data-overlay-theme	*letter* (a-z)	规定对话框页面的蒙版（背景）颜色
data-theme	*letter* (a-z)	规定对话框页面的主题颜色
data-title	*sometext*	规定对话框页面的标题

增强

带有 data-enhance="false" 或 data-ajax="false" 的容器。

Data-属性	值	描述
data-enhance	**true** | false	如果设置为 "true"（默认），jQuery Mobile 会自动渲染页面，使其更适合于移动设备。如果设置为 "false"，框架将不会渲染页面
data-ajax	**true** | false	规定是否通过 ajax 加载页面

注意：data-enhance="false" 必须与 $.mobile.ignoreContentEnabled=true" 一同使用来阻止 jQuery Mobile 自动渲染页面。

当 $.mobile.ignoreContentEnabled 设置为 true 时，data-ajax="false" 容器内的任何链接或表单元素将会被框架的导航功能忽略。

域容器

包围在 label/表单元素对周围的带有 data-role="fieldcontain" 的容器。

固定工具栏

带有 data-role="header" 或 data-role="footer"，并带有 data-position="fixed" 属性的容器。

Data-属性	值	描述
data-disable-page-zoom	**true** | false	规定用户是否能缩放页面
data-fullscreen	true | **false**	规定工具栏是否一直固定在顶部或底部
data-tap-toggle	**true** | false	规定用户是否能够通过点击改变工具栏的可见性
data-transition	**slide** | fade | none	规定当点击发生时的切换效果
data-update-page-padding	**true** | false	规定页面顶部和底部的内边距是否在 resize、transition 和 "updatelayout" 事件发生时更新（jQuery Mobile 在 "pageshow" 事件发生时总是更新内边距）
data-visible-on-page-show	**true** | false	规定当父页面显示时工具栏的可见性

翻转拨动开关

一个带有 data-role="slider" 的 <select> 元素和两个 <option> 元素。

Data-属性	值	描述
data-mini	true | **false**	规定开关是小尺寸还是常规尺寸
data-role	none	防止 jQuery Mobile 把拨动开关渲染成按钮样式
data-theme	*letter* (a-z)	规定拨动开关的主题颜色
data-track-theme	*letter* (a-z)	规定轨道的主题颜色

尾部栏

带有 data-role="footer" 的容器。

Data-属性	值	描述
data-id	*sometext*	规定唯一 ID。对于持续的尾部栏是必需的
data-position	**inline** | fixed	规定尾部栏是与页面内容内联还是保持固定在底部
data-fullscreen	true | **false**	规定尾部栏是固定在底部还是覆盖在页面内容上（查看范围更大）
data-theme	*letter* (a-z)	规定尾部栏的主题颜色。默认是 "a"

注意：如需启用全屏定位，请使用 data-position="fixed"，然后添加 data-fullscreen 属性到元素上。

头部栏

带有 data-role="header" 的容器。

Data-属性	值	描述
data-id	*sometext*	规定唯一 ID。对于持续的头部栏是必需的
data-position	**inline** | fixed	规定头部栏是与页面内容内联还是保持固定在顶部
data-fullscreen	true | **false**	规定头部栏是固定在顶部还是覆盖在页面内容上（查看范围更大）
data-theme	*letter* (a-z)	规定头部栏的主题颜色。默认是 "a"

注意：如需启用全屏定位，请使用 data-position="fixed"，然后添加 data-fullscreen 属性到元素上。

链接

所有的链接，包含那些带有 data-role="button" 的链接和表单提交按钮。

Data-属性	值	描述
data-ajax	**true** | false	规定是否通过 ajax 加载页面来提高用户体验和交互。如果设置为 false，jQuery Mobile 将会执行一个正常的页面请求。
data-direction	reverse	反向转换动画（仅用于页面和对话框）
data-dom-cache	true | **false**	规定是否清除各个页面的 jQuery DOM 缓存（如果设置为 true，您必须自己管理 DOM 并在所有移动设备上进行测试）
data-prefetch	true | **false**	规定是否预先读取页面到 DOM 中，以便当用户访问它们时可用
data-rel	back | dialog | external | popup	规定链接行为的选项。Back - 回退到历史记录中的前一个页面。Dialog - 以对话框形式打开链接，不保存到历史记录中。External - 用于链接到另一个域。Popup - 打开一个弹出窗口。
data-transition	**fade** | flip | flow | pop | slide | slidedown | slidefade | slideup | turn | none	规定一个页面切换到下一个页面的效果。请查看我们的 jQuery Mobile 页面切换章节。
data-position-to	**origin** | jQuery selector | window	规定弹出框的位置。Origin - 默认。定位弹窗在打开它的链接上。jQuery selector - 定位弹窗在指定元素上。Window - 定位弹窗在窗口屏幕的中央。

列表

带有 data-role="listview" 的 <ol> 或 。

Data-属性	值	描述
data-autodividers	true | **false**	规定是否自动划分列表项
data-count-theme	*letter* (a-z)	规定计数气泡的主题颜色。默认是 "c"
data-divider-theme	*letter* (a-z)	规定列表分隔的主题颜色。默认是 "b"
data-filter	true | **false**	规定是否在列表中添加搜索框
data-filter-placeholder	*sometext*	规定搜索框内的文本。默认是 "Filter items..."
data-filter-theme	*letter* (a-z)	规定搜索过滤的主题颜色。默认是 "c"
data-icon	Icons 参考手册	规定列表的图标
data-inset	true | **false**	规定列表是否渲染成圆角且带外边距
data-split-icon	Icons 参考手册	规定分割按钮的图表。默认是 "arrow-r"
data-split-theme	*letter* (a-z)	规定分割按钮的主题颜色。默认是 "b"
data-theme	*letter* (a-z)	规定列表的主题颜色

列表项

带有 data-role="listview" 的 <ol> 或 <ul> 内的 。

Data-属性	值	描述
data-filtertext	*sometext*	规定当过滤元素时要搜索的文本。该文本将会被过滤，而不是实际的列表项文本
data-icon	Icons 参考手册	规定列表项图标
data-role	list-divider	规定列表项的分隔
data-theme	*letter* (a-z)	规定列表项的主题颜色

注意：data-icon 属性只作用于带有链接的列表项。

导航栏

带有 data-role="navbar" 容器内部的 <li> 元素。

Data-属性	值	描述
data-icon	Icons 参考手册	规定列表项的图标
data-iconpos	left | right | **top** | bottom | notext	规定图标的位置

 导航栏从他们的父容器中继承了主题样本。为导航栏设置 data-theme 属性是不可能的。可以为导航栏中的每个链接单独设置 data-theme 属性。

页面

带有 data-role="page" 的容器。

Data-属性	值	描述
data-add-back-btn	true | **false**	自动添加后退按钮，仅限头部栏
data-back-btn-text	*sometext*	规定后退按钮的一些文本
data-back-btn-theme	*letter* (a-z)	规定后退按钮的主题颜色
data-close-btn-text	*letter* (a-z)	规定对话框上关闭按钮的一些文本
data-dom-cache	true | **false**	规定是否清除各个页面的 jQuery DOM 缓存（如果设置为 true，您必须自己管理 DOM 并在所有移动设备上进行测试）
data-overlay-theme	*letter* (a-z)	规定对话框页面的蒙版（背景）颜色
data-theme	*letter* (a-z)	规定页面的主题颜色。默认是 "c"
data-title	*sometext*	规定页面标题
data-url	url	更新 URL 的值，而不是用于请求页面的 URL

弹窗

带有 data-role="popup" 的容器。

Data-属性	值	描述
data-corners	**true** | false	规定弹窗是否圆角
data-overlay-theme	*letter* (a-z)	规定弹出框的蒙版（背景）颜色。默认是透明背景（无）
data-shadow	**true** | false	规定弹出框是否有阴影
data-theme	*letter* (a-z)	规定弹出框的主题颜色。默认是继承的，"none" 设置弹窗为透明的
data-tolerance	30, 15, 30, 15	规定窗口边缘（上 top、右 right、下 bottom、左 left）的距离

带有 data-rel="popup" 的锚：

Data-属性	值	描述
data-position-to	**origin** | jQuery selector | window	>规定弹出框的位置。Origin - 默认。定位弹窗在打开它的链接上。jQuery selector - 定位弹窗在指定元素上。Window - 定位弹窗在窗口屏幕的中央。
data-rel	popup	用于打开弹出框
data-transition	fade | flip | flow | pop | slide | slidedown | slidefade | slideup | turn | **none**	规定一个页面切换到下一个页面的效果。请查看我们的 jQuery Mobile 页面切换章节。

单选按钮

带有 type="radio" 的成双成对的 label 和 input。它们会被自动渲染程按钮样式，data-role 不是必需的。

Data-属性	值	描述
data-mini	true | **false**	规定按钮是小尺寸还是常规尺寸
data-role	none	防止 jQuery Mobile 渲染单选按钮的样式为增强状态的按钮，使元素以 HTML 原生的状态显示
data-theme	*letter* (a-z)	规定单选按钮的主题颜色

 如需组合多个单选按钮，请使用带有 data-role="controlgroup" 属性和 data-type="horizontal|vertical" 的容器来规定是否水平或垂直组合单选按钮。

选择

所有的 <select> 元素。这些会被自动渲染成按钮样式，date-role 是不必需的。

Data-属性	值	描述
data-icon	Icons 参考手册	规定 select 元素的图标。默认是 "arrow-d"
data-iconpos	left | **right** | top | bottom | notext	规定图标的位置
data-inline	true | **false**	规定 select 元素是否内联
data-mini	true | **false**	规定 select 元素是小尺寸还是常规尺寸
data-native-menu	**true** | false	当设置为 false 时，它使用 jQuery 自带的自定义的选择菜单（如果您想要让选择菜单在所有的移动设备上都显示相同，则推荐这么使用）
data-overlay-theme	*letter* (a-z)	规定 jQuery 自带的自定义的选择菜单的主题颜色（与 data-native-menu="false" 一起使用）
data-placeholder	true | **false**	可在一个非原生的选择菜单的 <option> 元素上设置
data-role	none	防止 jQuery Mobile 把 select 元素渲染成按钮样式
data-theme	*letter* (a-z)	规定 select 元素的主题颜色

>  如需组合多个 select 元素，请使用带有 data-role="controlgroup" 属性和 data-type="horizontal|vertical" 的容器来规定是否水平或垂直组合 select 元素。

滑动块

带有 type="range" 的输入框。这些会被自动渲染成按钮样式，date-role 是不必需的。

Data-属性	值	描述
data-highlight	true | **false**	规定滑动轨道是否高亮突出显示
data-mini	true | **false**	规定滑动块是小尺寸还是常规尺寸
data-role	none	防止 jQuery Mobile 渲染滑动块控件为按钮样式
data-theme	*letter* (a-z)	规定滑动块控件（输入框、手柄和轨道）的主题颜色
data-track-theme	*letter* (a-z)	规定滑动块轨道的主题颜色

文本输入框 & 文本输入域

带有 type="text|search|etc." 的 input 或 textarea 元素。这些会被自动渲染成按钮样式，date-role 是不必需的。

Data-属性	值	描述
data-mini	true | **false**	规定输入框是小尺寸还是常规尺寸
data-role	none	防止 jQuery Mobile 把输入框/输入域渲染成按钮样式
data-theme	*letter* (a-z)	规定输入字段的主题颜色

jQuery Mobile 图标

jQuery 图标

在 jQuery Mobile 中，如需为按钮添加图标，请使用 data-icon 属性：

```
<a href="#anylink" data-role="button" **data-icon="refresh"**>Refre
```

提示：在 <button> 或 <input> 元素中，您也可以使用 data-icon 属性。

下面我们列出了 jQuery Mobile 提供的所有可用图标：

属性值	描述	图标	实例
data-icon="arrow-l"	左箭头		
data-icon="arrow-r"	右箭头		
data-icon="arrow-u"	上箭头		
data-icon="arrow-d"	下箭头		
data-icon="plus"	加号		
data-icon="minus"	减号		
data-icon="delete"	删除		
data-icon="check"	检查		
data-icon="home"	首页		
data-icon="info"	信息		
data-icon="grid"	网格		
data-icon="gear"	工具		
data-icon="search"	搜索		
data-icon="back"	后退		
data-icon="forward"	前进		
data-icon="refresh"	更新		
data-icon="star"	星号		
data-icon="alert"	警告		

jQuery Mobile 事件

jQuery Mobile 事件参考手册

以下列表为所有的 jQuery Mobile 事件。

注意：请使用 on() 方法绑定事件。

事件	描述
hashchange	启用可标记 #hash 历史，哈希值会在一次独立的点击时发生时变化，比如一个用户点击后退按钮，会通过hashchange事件进行处理。
navigate	包裹了 hashchange 和 popstate 的事件
orientationchange	方向改变事件,在用户垂直或者水平旋转移动设备时触发。
pagebeforechange	在页面切换之前，触发的事件。使用$.mobile.changePage()来切换页面，此方法触发2个事件，切换之前的pagebeforechange事件，和切换完成后pagechange（成功）或者pagechangefailed（失败）。
pagebeforecreate	页面初始化时，初始化之前触发。
pagebeforehide	在页面切换后旧页面隐藏之前，触发的事件。
pagebeforeload	在加载请求发出之前触发
pagebeforeshow	在页面切换后显示之前，触发的事件。
pagechange	在页面切换成功后，触发的事件。使用$.mobile.changePage()来切换页面，此方法触发2个事件，切换之前的pagebeforechange事件，和切换完成后pagechange（成功）或者pagechangefailed（失败）。
pagechangefailed	在页面切换失败时，触发的事件。使用$.mobile.changePage()来切换页面，此方法触发2个事件，切换之前的pagebeforechange事件，和切换完成后pagechange（成功）或者pagechangefailed（失败）。
pagecreate	在页面创建成功之后，触发的事件,但增强完成之前。
pagehide	在页面切换后老页面隐藏之后，触发的事件。
pageinit	在页面页面初始化时，触发的事件。
pageload	在页面完全加载成功后触发。

pageloadfailed	如果页面请求失败触发。
pageremove	在窗口视图从 DOM 中移除外部页面之前触发。
pageshow	在过渡动画完成后，在"到达"页面触发。
scrollstart	当用户开始滚动页面时触发。
scrollstop	当用户停止滚动页面时触发。
swipe	当用户在元素上水平滑动时触发。
swipeleft	当用户从左划过元素超过 30px 时触发。
swiperight	当用户从右划过元素超过 30px 时触发。
tap	当用户敲击某元素时触发。
taphold	当元素敲击某元素并保持一秒时触发。
throttledresize	启用可标记 #hash 历史记录
updatelayout	由动态显示/隐藏内容的 jQuery Mobile 组件触发。
vclick	虚拟化的 click 事件处理器
vmousecancel	虚拟化的 mousecancel 事件处理器
vmousedown	虚拟化的 mousedown 事件处理器
vmousemove	虚拟化的 mousemove 事件处理器
vmouseout	虚拟化的 mouseout 事件处理器
vmouseover	虚拟化的 mouseover 事件处理器
vmouseup	虚拟化的 mouseup 事件处理器

jQuery Mobile 页面事件

jQuery Mobile 页面事件

在 jQuery Mobile 中与页面打交道的事件被分为四类：

- Page Initialization - 在页面创建前，当页面创建时，以及在页面初始化之后
- Page Load/Unload - 当外部页面加载时、卸载时或遭遇失败时
- Page Transition - 在页面过渡之前和之后
- Page Change - 当页面被更改，或遭遇失败时

如需关于所有 jQuery Mobile 事件的完整信息，请访问我们的 jQuery Mobile 事件参考手册。

jQuery Mobile Initialization 事件

当 jQuery Mobile 中的一张典型页面进行初始化时，它会经历三个阶段：

- 在页面创建前
- 页面创建
- 页面初始化

每个阶段触发的事件都可用于插入或操作代码。

事件	描述
pagebeforecreate	当页面即将初始化，并且在 jQuery Mobile 已开始增强页面之前，触发该事件。
pagecreate	当页面已创建，但增强完成之前，触发该事件。
pageinit	当页面已初始化，并且在 jQuery Mobile 已完成页面增强之后，触发该事件。

下面的例子演示在 jQuery Mobile 中创建页面时，何时触发每种事件：

实例

```
$(document).on("pagebeforecreate",function(event){
  alert("触发 pagebeforecreate事件！");
});
$(document).on("pagecreate",function(event){
  alert("触发 pagecreate 事件！");
});
$(document).on("pageinit",function(event){
  alert("触发 pageinit 事件！")
});
```

jQuery Mobile Load 事件

页面加载事件属于外部页面。

无论外部页面何时载入 DOM，将触发两个事件。第一个是 pagebeforeload，第二个是 pageload （成功）或 pageloadfailed（失败）。

下表中解释了这些事件：

事件	描述
pagebeforeload	在任何页面加载请求作出之前触发。
pageload	在页面已成功加载并插入 DOM 后触发。
pageloadfailed	如果页面加载请求失败，则触发该事件。默认地，将显示 "Error Loading Page" 消息。

下列演示 pageload 和 pagloadfailed 事件的工作原理：

实例

```
$(document).on("pageload",function(event,data){
  alert("触发 pageload 事件！\nURL: " + data.url);
});
$(document).on("pageloadfailed",function(event,data){
  alert(";抱歉，被请求页面不存在。");
});
```

jQuery Mobile 过渡事件

我们还可以在从一页过渡到下一页时使用事件。

页面过渡涉及两个页面：一张"来"的页面和一张"去"的页面 - 这些过渡使当前活动页面（"来的"页面）到新页面（"去的"页面的改变过程变得更加动感。

事件	描述
pagebeforeshow	在"去的"页面触发，在过渡动画开始前。
pageshow	在"去的"页面触发，在过渡动画完成后。
pagebeforehide	在"来的"页面触发，在过渡动画开始前。
pagehide	在"来的"页面触发，在过渡动画完成后。

下列演示了过渡时间的工作原理：

实例

```
$(document).on("pagebeforeshow","#pagetwo",function(){ //当进入页面二
  alert("页面二即将显示");
});
$(document).on("pageshow","#pagetwo",function(){ // 当进入页面二时
  alert("现在显示页面二");
});
$(document).on("pagebeforehide","#pagetwo",function(){ // 当进入页面
  alert("页面二即将隐藏");
});
$(document).on("pagehide","#pagetwo",function(){ // When leaving pa
  alert("现在隐藏页面二");
});
```

jQuery Mobile CSS 类

jQuery CSS 类

jQuery Mobile CSS 类来设置不同元素的样式。

以下列表包含了通用的 CSS 样式：

全局类

以下类可以在 jQuery Mobile 小工具中使用 (按钮，工具条，面板，表格，列表等。):

Class	描述
ui-corner-all	为元素添加圆角
ui-shadow	为元素添加阴影
ui-overlay-shadow	为元素添加多层阴影
ui-mini	让元素变小

按钮类

除了全局类外，你可以向 <a> 或 <button> 元素添加以下类 (不是 <input> 按钮):

Class	描述
ui-btn	添加在 <a> 元素上并以按钮形式展示
ui-btn-inline	在同一行上显示按钮
ui-btn-icon-top	定位图标在按钮文本之上
ui-btn-icon-right	定位图标在按钮文本的右边
ui-btn-icon-bottom	定位图标在按钮文本之下
ui-btn-icon-left	定位图标在按钮文本的左边
ui-btn-icon-notext	只显示图标
ui-btn-a|b	指定按钮演示。"a" 是默认的 (灰色背景黑色文本样式), "b" 修改颜色为黑色背景白色文本

图标类

所有可用图片的类用在 <a> 和 <button> 元素上 (查看 jQuery Mobile 图标参考手册了解如何在其他元素上使用):

Class	图标描述	图标
ui-icon-action	动作 (一般用于页面跳转切换)	
ui-icon-alert	三角形内的感叹号	
ui-icon-audio	音响/音箱	
ui-icon-arrow-d-l	左下角箭头	
ui-icon-arrow-d-r	右下角箭头	
ui-icon-arrow-u-l	左上角箭头	
ui-icon-arrow-u-r	右上角箭头	
ui-icon-arrow-l	左角箭头	
ui-icon-arrow-r	右角箭头	

ui-icon-arrow-u	向上箭头	
ui-icon-arrow-d	向下箭头	
ui-icon-back	返回	
ui-icon-bars	三条水平线，一般用于展示列表按钮图标	
ui-icon-bullets	用于展示列表按钮图标	
ui-icon-calendar	日历	
ui-icon-camera	相机	
ui-icon-carat-d	向下滑动图标	
ui-icon-carat-l	向左滑动图标	
ui-icon-carat-r	向右滑动图标	
ui-icon-carat-u	向上滑动图标	
ui-icon-check	勾选	
ui-icon-clock	闹钟	
ui-icon-cloud	云	
ui-icon-comment	评论 / 消息	
ui-icon-delete	删除	
ui-icon-edit	编辑 / 铅笔	
ui-icon-eye	眼睛	
ui-icon-forbidden	禁用标识 sign	
ui-icon-forward	撤销 - (返回上一步)	
ui-icon-gear	齿轮，一般用于设置按钮图标	
ui-icon-grid	网格	
ui-icon-heart	心型，可用于文章收藏	

ui-icon-home	主页	
ui-icon-info	信息	
ui-icon-location	定位	
ui-icon-lock	锁	
ui-icon-mail	邮件 / 信封	
ui-icon-minus	减号	
ui-icon-navigation	导航	
ui-icon-phone	电话	
ui-icon-power	开关 (On/off)	
ui-icon-plus	加号	
ui-icon-recycle	循环 标识	
ui-icon-refresh	刷新	
ui-icon-search	搜索 / 放大镜	
ui-icon-shop	商店/购物袋	
ui-icon-star	星号	
ui-icon-tag	标签	
ui-icon-user	用户 / 人物	
ui-icon-video	视频 / 相机	

主题类 Classes

jQuery Mobile 提供了两个主题类: a (灰) 和 b (黑)。 但是，你可以创建你自己的自定义类 - 可定义到字母 "z" （查看 jQuery Mobile 主题 ）。 下表列出了可用的主题类。 字母 (a-z) 意为样式可以指定 a 到 z。例如 ui-bar-a 或 ui-bar-b等。

Class	描述
ui-bar-(a-z)	指定栏目演示 - 头部，底部及其他栏目
ui-body-(a-z)	指定内容块的颜色 - 页面内容部分 (1.4.0 版本已废弃), 列表视图， 弹窗, 侧栏， 面板， 加载， 折叠。
ui-btn-(a-z)	指定按钮颜色
ui-group-theme-(a-z)	定义了控制组的演示 listviews 和 collapsible 集合。
ui-overlay-(a-z)	定义了页面背景颜色，包括对话框， 弹窗和其他出现在最顶层的页面容器
ui-page-theme-(a-z)	定义了页面演示

网格类

网格中的列是等宽的（合计是 100%），没有边框、背景、margin 或 padding。 这里有四种布局网格可供使用：

网格类	列	列宽	对应	实例
ui-grid-solo	1	100%	ui-block-a	
ui-grid-a	2	50% / 50%	ui-block-a|b	
ui-grid-b	3	33% / 33% / 33%	ui-block-a|b|c	
ui-grid-c	4	25% / 25% / 25% / 25%	ui-block-a|b|c|d	
ui-grid-d	5	20% / 20% / 20% / 20% / 20%	ui-block-a|b|c|d|e	

更多信息可以查看 jQuery Mobile 网格 章节。

W3School Ionic 教程

作者：W3School

来源：Ionic 教程

ionic 入门

ionic 简介



ionic 是一个强大的 HTML5 应用程序开发框架(HTML5 Hybrid Mobile App Framework)。 可以帮助您使用 Web 技术，比如 HTML、CSS 和 Javascript 构建接近原生体验的移动应用程序。

ionic 主要关注外观和体验，以及和你的应用程序的 UI 交互，特别适合用于基于 Hybird 模式的 HTML5 移动应用程序开发。

ionic是一个轻量的手机UI库，具有速度快，界面现代化、美观等特点。为了解决其他一些UI库在手机上运行缓慢的问题，它直接放弃了IOS6和Android4.1以下的版本支持，来获取更好的使用体验。

ionic 特点

- 1.ionic 基于Angular语法，简单易学。
- 2.ionic 是一个轻量级框架。
- 3.ionic 完美的融合下一代移动框架，支持 Angularjs 的特性， MVC ，代码易维护。
- 4.ionic 提供了漂亮的设计，通过 SASS 构建应用程序，它提供了很多 UI 组件来帮助开发者开发强大的应用。
- 5.ionic 专注原生，让你看不出混合应用和原生的区别
- 6.ionic 提供了强大的命令行工具。
- 7.ionic 性能优越，运行速度快。

学习本教程前你需要了解？

学习本教程前你需要了解以下基础知识：

- HTML
- CSS
- Javascript
- Angular

ionic 相关内容

ionic 官方网站：http://ionicframework.com/

ionic 官方文档：http://ionicframework.com/docs/

Github 地址：https://github.com/driftyco/ionic

ionic 安装

本站实例采用了ionic v1.0.1 版本，下载地址为：ionic-v1.0.1.zip。

ionic 最新版本下载地址：http://ionicframework.com/docs/overview/#download。

下载后解压压缩包，包含以下目录：

```
css/              =>     样式文件
fonts/            =>     字体文件
js/               =>     Javascript文件
version.json      =>     版本更新说明
```

你也可以在 Github 上下载以下资源文件：https://github.com/driftyco/ionic（在 release 目录中）。

接下来，我们只需要在项目中引入以上目录中的 css/ionic.min.css 和 js/ionic.bundle.min.js 文件即可创建 ionic 应用。

实例

```
<ion-header-bar class="bar-positive">
    <h1 class="title">Hello World!</h1>
</ion-header-bar>

<ion-content>
    <p>我的第一个 ionic 应用。</p>
</ion-content>
```

点击 "尝试一下" 按钮查看在线实例。

本教程着重讲解 ionic 框架的应用，大部分实例在浏览器中运行，移动设备上运行可以在接下来的命令行安装教程中详细了解。

注意：在移动应用如 phonegap 中我们只需将对应的 js 和 css 文件加入到资源库中即可。

命令行安装

首先您需要安装 Node.js，我们在接下来的安装中需要使用到其 NPM 工具，更多 NPM 介绍可以查看我们的NPM 使用介绍。

然后通过命令行工具安装最新版本的 cordova 和 ionic 。通过参考Android 和 iOS 官方文档来安装。

Window 和 Linux 上打开命令行工具执行以下命令：

```
$ npm install -g cordova ionic
```

Mac 系统上使用以下命令：

```
sudo npm install -g cordova ionic
```

提示*: IOS*需要在*Mac Os X.* 和*Xcode*环境下面安装使用。

如果你已经安装了以上环境，可以执行以下命令来更新版本:

```
npm update -g cordova ionic
```

或

```
sudo npm update -g cordova ionic
```

创建应用

使用ionic官方提供的现成的应用程序模板，或一个空白的项目创建一个ionic应用：

```
$ ionic start myApp tabs
```

运行我们刚才创建的**ionic**项目

使用 ionic tool 创建，测试，运行你的apps(或者通过Cordova直接创建)。

创建android应用:

```
$ cd myApp
$ ionic platform add android
$ ionic build android
$ ionic emulate android
```

创建ios应用:

```
$ cd myApp
$ ionic platform add ios
$ ionic build ios
$ ionic emulate ios
```

ionic 创建 APP

前面的章节中我们已经学会了 ionic 框架如何导入到项目中。

接下来我们将为大家介绍如何创建一个 ionic APP 应用。

ionic 创建 APP 使用 HTML、CSS 和 Javascript 来构建，所以我们可以创建一个 www 目录，并在目录下创建 index.html 文件，代码如下：

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Todo</title>
    <meta name="viewport" content="initial-scale=1, maximum-scale=1

    <link href="lib/ionic/css/ionic.css" rel="stylesheet">

    <script src="lib/ionic/js/ionic.bundle.js"></script>

    <!-- 在使用 Cordova/PhoneGap 创建的 APP 中包含的文件，由 Cordova/Ph
    <script src="cordova.js"></script>
  </head>
  <body>
  </body>
</html>
```

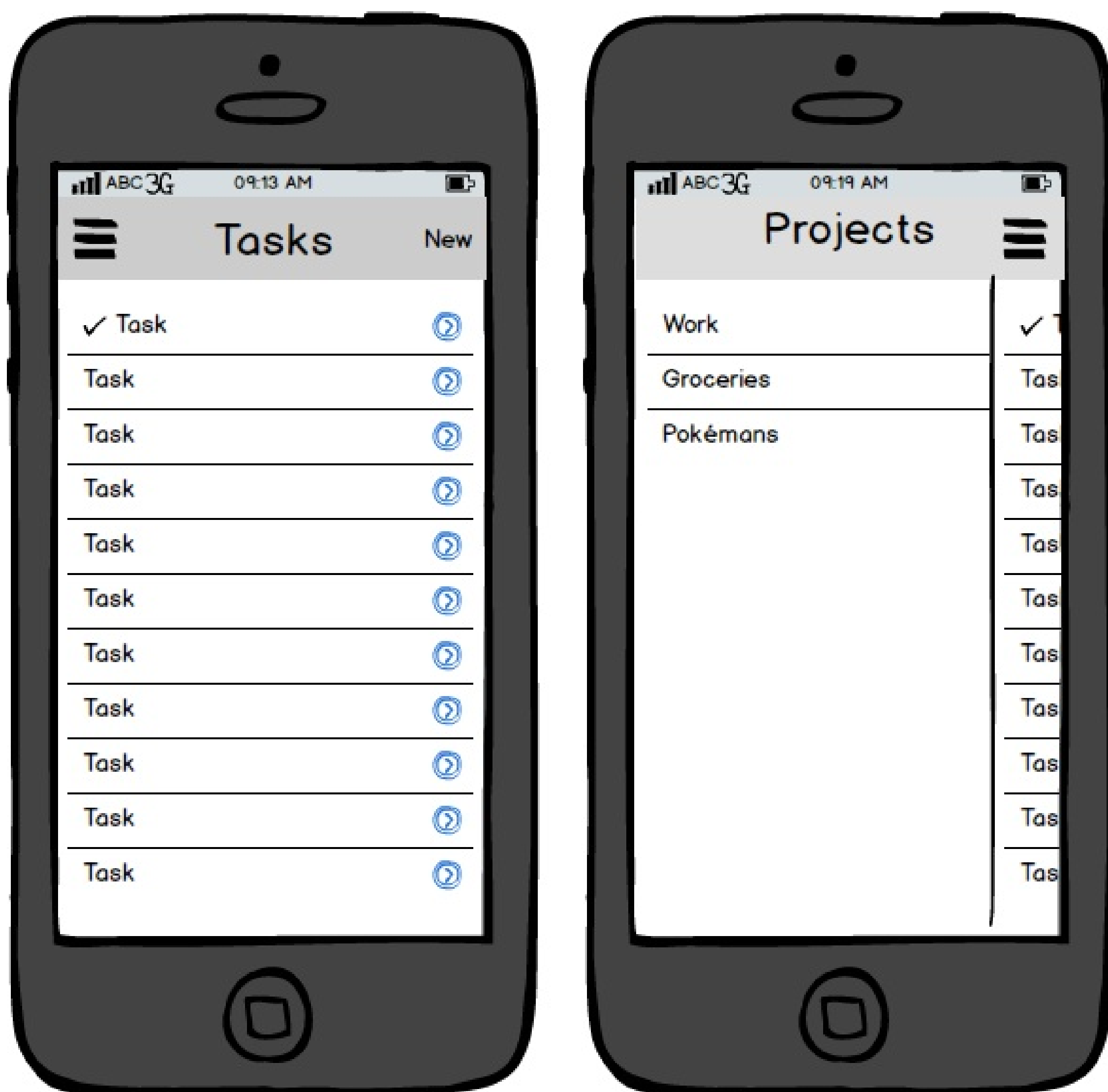
以上代码中，我们引入了 Ionic CSS 文件、 Ionic JS 文件及 Ionic AngularJS 扩展 ionic.bundle.js（ionic.bundle.js）。

ionic.bundle.js 文件已经包含了 Ionic核心JS、AngularJS、Ionic的AngularJS扩展，如果你需要引入其他 Angular 模块，可以从 lib/js/angular 目录中调用。

cordova.js 是在使用 Cordova/PhoneGap 创建应用时生成的，不需要手动引入，你可以在 Cordova/PhoneGap 项目中找到该文件，所以在开发过程中显示 404 是正常的。

创建 APP

接下来我们来实现一个包含标题、侧边栏、列表等的应用,设计图如下：



创建侧边栏

侧边栏创建使用 ion-side-menus 控制器。

编辑我们先前创建的 index.html 文件，修改 <body> 内的内容，如下所示：

```
<body>
  <ion-side-menus>
    <ion-side-menu-content>
    </ion-side-menu-content>
    <ion-side-menu side="left">
    </ion-side-menu>
  </ion-side-menus>
</body>
```

控制器解析：

- **ion-side-menus**： 是一个带有边栏菜单的容器，可以通过点击按钮或者滑动屏幕来展开菜单。

- **ion-side-menu-content**：展示主要内容的容器，可以通过滑动屏幕来展开 menu。
- **ion-side-menu**：存放侧边栏的容器。

初始化 APP

接下来我们创建一个新的 AngularJS 模块，并初始化，代码位于 www/js/app.js 中：

```
angular.module('todo', ['ionic'])
```

之后在我们的 body 标签中添加 ng-app 属性：

```
<body ng-app="todo">
```

在 index.html 文件的 <script src="cordova.js"></script> 上面引入 app.js 文件：

```
<script src="js/app.js"></script>
```

修改 index.html 文件 body 标签的内容，代码如下所示：

```
<body ng-app="todo">
  <ion-side-menus>

    <!-- 中心内容 -->
    <ion-side-menu-content>
      <ion-header-bar class="bar-dark">
        <h1 class="title">Todo</h1>
      </ion-header-bar>
      <ion-content>
      </ion-content>
    </ion-side-menu-content>

    <!-- 左侧菜单 -->
    <ion-side-menu side="left">
      <ion-header-bar class="bar-dark">
        <h1 class="title">Projects</h1>
      </ion-header-bar>
    </ion-side-menu>

  </ion-side-menus>
</body>
```

以上步骤我们已经完成了 ionic 基本 APP 的应用。

创建列表

首先创建一个控制器 **TodoCtrl**：

```
<body ng-app="todo" ng-controller="TodoCtrl">
```

在app.js文件中，使用控制器定义列表数据：

```
angular.module('todo', ['ionic'])

.controller('TodoCtrl', function($scope) {
  $scope.tasks = [
    { title: '菜鸟教程' },
    { title: 'www.runoob.com' },
    { title: '菜鸟教程' },
    { title: 'www.runoob.com' }
  ];
});
```

在index.html页面中数据列表我们使用 Angular ng-repeat 来迭代数据：

```
<!-- 中心内容 -->
<ion-side-menu-content>
  <ion-header-bar class="bar-dark">
    <h1 class="title">Todo</h1>
  </ion-header-bar>
  <ion-content>
    <!-- 列表 -->
    <ion-list>
      <ion-item ng-repeat="task in tasks">
        {{task.title}}
      </ion-item>
    </ion-list>
  </ion-content>
</ion-side-menu-content>
```

修改后 index.html body 标签内代码如下：

```
<body ng-app="todo" ng-controller="TodoCtrl">
    <ion-side-menus>

    <!-- 中心内容 -->
    <ion-side-menu-content>
      <ion-header-bar class="bar-dark">
        <h1 class="title">Todo</h1>
      </ion-header-bar>
      <ion-content>
        <!-- 列表 -->
        <ion-list>
          <ion-item ng-repeat="task in tasks">
            {{task.title}}
          </ion-item>
        </ion-list>
      </ion-content>
    </ion-side-menu-content>

    <!-- 左侧菜单 -->
    <ion-side-menu side="left">
      <ion-header-bar class="bar-dark">
        <h1 class="title">Projects</h1>
      </ion-header-bar>
    </ion-side-menu>

    </ion-side-menus>
    <script src="http://www.runoob.com/static/ionic/js/app.js"></sc
    <script src="cordova.js"></script>
</body>
```

创建添加页面

修改 index.html 在 **</ion-side-menus>** 后添加如下代码:

```
<script id="new-task.html" type="text/ng-template">

  <div class="modal">

    <!-- Modal header bar -->
    <ion-header-bar class="bar-secondary">
      <h1 class="title">New Task</h1>
      <button class="button button-clear button-positive" ng-click=
    </ion-header-bar>

    <!-- Modal content area -->
    <ion-content>

      <form ng-submit="createTask(task)">
        <div class="list">
          <label class="item item-input">
            <input type="text" placeholder="What do you need to do?
          </label>
        </div>
        <div class="padding">
          <button type="submit" class="button button-block button-p
        </div>
      </form>

    </ion-content>

  </div>

</script>
```

以上代码中我们通过 **<script id="new-task.html" type="text/ng-template">** 定义了新的模板页面。

表单提交时触发 createTask(task) 函数。

ng-model="task.title" 会将表单的输入数据赋值给 task 对象的 title 属性。

修改 **<ion-side-menu-content>** 内的内容，代码如下：

```
<!-- 中心内容 -->
<ion-side-menu-content>
<ion-header-bar class="bar-dark">
  <h1 class="title">Todo</h1>
  <!-- 新增按钮-->
  <button class="button button-icon" ng-click="newTask()">
    <i class="icon ion-compose"></i>
  </button>
</ion-header-bar>
<ion-content>
  <!-- 列表 -->
  <ion-list>
    <ion-item ng-repeat="task in tasks">
      {{task.title}}
    </ion-item>
  </ion-list>
</ion-content>
</ion-side-menu-content>
```

app.js 文件中，控制器代码如下：

```
angular.module('todo', ['ionic'])

.controller('TodoCtrl', function($scope, $ionicModal) {
  $scope.tasks = [
    { title: '菜鸟教程' },
    { title: 'www.runoob.com' },
    { title: '菜鸟教程' },
    { title: 'www.runoob.com' }
  ];

  // 创建并载入模型
  $ionicModal.fromTemplateUrl('new-task.html', function(modal) {
    $scope.taskModal = modal;
  }, {
    scope: $scope,
    animation: 'slide-in-up'
  });

  // 表单提交时调用
  $scope.createTask = function(task) {
    $scope.tasks.push({
      title: task.title
    });
    $scope.taskModal.hide();
    task.title = "";
  };

  // 打开新增的模型
  $scope.newTask = function() {
    $scope.taskModal.show();
  };

  // 关闭新增的模型
  $scope.closeNewTask = function() {
    $scope.taskModal.hide();
  };
});
```

创建侧边栏

通过以上步骤我们已经实现了新增功能，现在我们为 app 添加侧边栏功能。

修改 内的内容，代码如下：

```
<!-- 中心内容 -->
<ion-side-menu-content>
  <ion-header-bar class="bar-dark">
    <button class="button button-icon" ng-click="toggleProjects()">
      <i class="icon ion-navicon"></i>
    </button>
    <h1 class="title">{{activeProject.title}}</h1>
    <!-- 新增按钮 -->
    <button class="button button-icon" ng-click="newTask()">
      <i class="icon ion-compose"></i>
    </button>
  </ion-header-bar>
  <ion-content scroll="false">
    <ion-list>
      <ion-item ng-repeat="task in activeProject.tasks">
        {{task.title}}
      </ion-item>
    </ion-list>
  </ion-content>
</ion-side-menu-content>
```

添加侧边栏：

```
<!-- 左边栏 -->
<ion-side-menu side="left">
<ion-header-bar class="bar-dark">
  <h1 class="title">Projects</h1>
  <button class="button button-icon ion-plus" ng-click="newProject(
  </button>
</ion-header-bar>
<ion-content scroll="false">
  <ion-list>
    <ion-item ng-repeat="project in projects" ng-click="selectProje
      {{project.title}}
    </ion-item>
  </ion-list>
</ion-content>
</ion-side-menu>
```

<ion-item> 中的 ng-class 指令设置菜单激活样式。

这里我们创建一个新的js 文件 app2.js(为了不与前面的混淆)，代码如下：

```
angular.module('todo', ['ionic'])
/**
 * The Projects factory handles saving and loading projects
 * from local storage, and also lets us save and load the
```

```
 * last active project index.
 */
.factory('Projects', function() {
  return {
    all: function() {
      var projectString = window.localStorage['projects'];
      if(projectString) {
        return angular.fromJson(projectString);
      }
      return [];
    },
    save: function(projects) {
      window.localStorage['projects'] = angular.toJson(projects);
    },
    newProject: function(projectTitle) {
      // Add a new project
      return {
        title: projectTitle,
        tasks: []
      };
    },
    getLastActiveIndex: function() {
      return parseInt(window.localStorage['lastActiveProject']) ||
    },
    setLastActiveIndex: function(index) {
      window.localStorage['lastActiveProject'] = index;
    }
  }
})

.controller('TodoCtrl', function($scope, $timeout, $ionicModal, Pro

  // A utility function for creating a new project
  // with the given projectTitle
  var createProject = function(projectTitle) {
    var newProject = Projects.newProject(projectTitle);
    $scope.projects.push(newProject);
    Projects.save($scope.projects);
    $scope.selectProject(newProject, $scope.projects.length-1);
  }

  // Load or initialize projects
  $scope.projects = Projects.all();

  // Grab the last active, or the first project
  $scope.activeProject = $scope.projects[Projects.getLastActiveInde

  // Called to create a new project
  $scope.newProject = function() {
    var projectTitle = prompt('Project name');
    if(projectTitle) {
      createProject(projectTitle);
    }
```

```
  };

  // Called to select the given project
  $scope.selectProject = function(project, index) {
    $scope.activeProject = project;
    Projects.setLastActiveIndex(index);
    $ionicSideMenuDelegate.toggleLeft(false);
  };

  // Create our modal
  $ionicModal.fromTemplateUrl('new-task.html', function(modal) {
    $scope.taskModal = modal;
  }, {
    scope: $scope
  });

  $scope.createTask = function(task) {
    if(!$scope.activeProject || !task) {
      return;
    }
    $scope.activeProject.tasks.push({
      title: task.title
    });
    $scope.taskModal.hide();

    // Inefficient, but save all the projects
    Projects.save($scope.projects);

    task.title = "";
  };

  $scope.newTask = function() {
    $scope.taskModal.show();
  };

  $scope.closeNewTask = function() {
    $scope.taskModal.hide();
  }

  $scope.toggleProjects = function() {
    $ionicSideMenuDelegate.toggleLeft();
  };

  // Try to create the first project, make sure to defer
  // this by using $timeout so everything is initialized
  // properly
  $timeout(function() {
    if($scope.projects.length == 0) {
      while(true) {
        var projectTitle = prompt('Your first project title:');
        if(projectTitle) {
          createProject(projectTitle);
          break;
```

```
        }
      }
    }
  });

});
```

body 中 ion-side-menus 代码如下：：

```
<ion-side-menus>

<!-- 中心内容 -->
<ion-side-menu-content>
  <ion-header-bar class="bar-dark">
    <button class="button button-icon" ng-click="toggleProjects()">
      <i class="icon ion-navicon"></i>
    </button>
    <h1 class="title">{{activeProject.title}}</h1>
    <!-- 新增按钮 -->
    <button class="button button-icon" ng-click="newTask()">
      <i class="icon ion-compose"></i>
    </button>
  </ion-header-bar>
  <ion-content scroll="false">
    <ion-list>
      <ion-item ng-repeat="task in activeProject.tasks">
        {{task.title}}
      </ion-item>
    </ion-list>
  </ion-content>
</ion-side-menu-content>

<!-- 左边栏 -->
<ion-side-menu side="left">
<ion-header-bar class="bar-dark">
  <h1 class="title">Projects</h1>
  <button class="button button-icon ion-plus" ng-click="newProject(
  </button>
</ion-header-bar>
<ion-content scroll="false">
  <ion-list>
    <ion-item ng-repeat="project in projects" ng-click="selectProje
      {{project.title}}
    </ion-item>
  </ion-list>
</ion-content>
</ion-side-menu>

</ion-side-menus>
```

ionic CSS

ionic 头部与底部

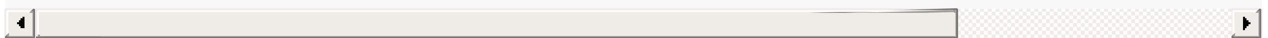
Header(头部)

Header是固定在屏幕顶部的组件,可以包如标题和左右的功能按钮。

ionic 默认提供了许多种颜色样式，你可以调用不同的样式名，当然也可以自定义一个。

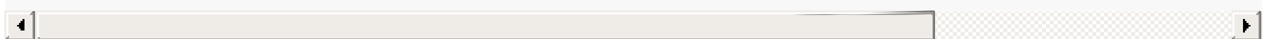
bar-light

```
<div  class="bar bar-header bar-light">  <h1  class="title">bar-lig
```



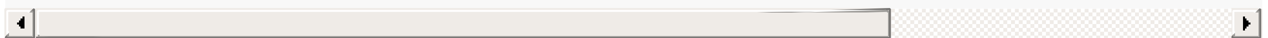
bar-stable

```
<div  class="bar bar-header bar-stable">  <h1  class="title">bar-st
```



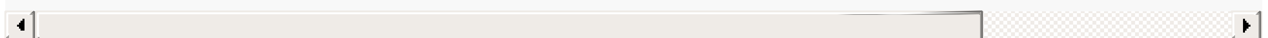
bar-positive

```
<div  class="bar bar-header bar-positive">  <h1  class="title">bar-
```



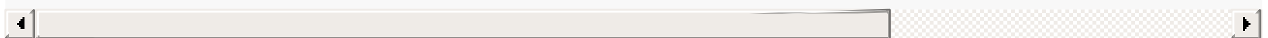
bar-calm

```
<div  class="bar bar-header bar-calm">  <h1  class="title">bar-calm
```



bar-balanced

```
<div  class="bar bar-header bar-balanced">  <h1  class="title">bar-
```



bar-energized

```
<div class="bar bar-header bar-energized"> <h1 class="title">bar
```

bar-assertive

```
<div class="bar bar-header bar-assertive"> <h1 class="title">bar
```

bar-royal

```
<div class="bar bar-header bar-royal"> <h1 class="title">bar-roy
```

bar-dark

```
<div class="bar bar-header bar-dark"> <h1 class="title">bar-dark
```

Sub Header（副标题）

Sub Header同样是固定在顶部，只是是在Header的下面，就算没有写Header这个，Sub Header这个样式也会距离顶部有一个Header的距离。颜色样式同 Header 。

```
<div class="bar bar-header"> <h1 class="title">Header</h1> </di
```

Footer(底部)

Footer 是在屏幕的最下方，可以包含多种内容类型。

```
<div class="bar bar-footer bar-balanced"> <div class="title">Foo
```

Footer 同上面的 Header，只是把样式名 bar-header 换做 bar-footer 。

```
<div  class="bar bar-footer">  <button  class="button button-clear'
```

此外，如果底部没有标题，但是又需要右边的按钮，你需要在右侧按钮添加 pull-right如:

```
<div  class="bar bar-footer">  <button  class="button button-clear
```

ionic 按钮

按钮是移动app不可或缺的一部分，不同风格的app，需要的不同按钮的样式。

默认情况下，按钮显示样式为：**display: inline-block**。

```
<button class="button">
  Default
</button>

<button class="button button-light">
  button-light
</button>

<button class="button button-stable">
  button-stable
</button>

<button class="button button-positive">
  button-positive
</button>

<button class="button button-calm">
  button-calm
</button>

<button class="button button-balanced">
  button-balanced
</button>

<button class="button button-energized">
  button-energized
</button>

<button class="button button-assertive">
  button-assertive
</button>

<button class="button button-royal">
  button-royal
</button>

<button class="button button-dark">
  button-dark
</button>
```

button-block 样式按钮显示为：**display: block**，它将完全填充父元素的宽度，包含了内边距属性padding。

```
<button class="button button-block button-positive">
  Block Button
</button>
```

使用 button-full 类，可以让按钮显示完全宽度，且不包含内边距padding。

```
<button class="button button-full button-positive">
  Full Width Block Button
</button>
```

不同大小的按钮

button-large 设置为大按钮，button-small 设置为小按钮。

```
<button class="button button-small button-assertive">
  Small Button
</button>
<button class="button button-large button-positive">
  Large Button
</button>
```

无背景按钮

button-outline 设置背景为透明。

```
<button class="button button-outline button-positive">
  Outlined Button
</button>
```

无背景与边框按钮

button-clear 设置按钮背景为透明，且无边框。

```
<button class="button button-clear button-positive">
  Clear Button
</button>
```

图标按钮

我们可以在按钮上添加图标。

```
<button class="button">
  <i class="icon ion-loading-c"></i> Loading...
</button>

<button class="button icon-left ion-home">Home</button>

<button class="button icon-left ion-star button-positive">Favorites

<a class="button icon-right ion-chevron-right button-calm">Learn Mo

<a class="button icon-left ion-chevron-left button-clear button-dar

<button class="button icon ion-gear-a"></button>

<a class="button button-icon icon ion-settings"></a>

<a class="button button-outline icon-right ion-navicon button-balar
```

头部/底部添加按钮

头部/底部可以添加按钮，按钮的样式根据头部/底部来设定，所以你不需要为按钮添加额外的样式。

```
<div class="bar bar-header">
  <button class="button icon ion-navicon"></button>
  <h1 class="title">Header Buttons</h1>
  <button class="button">Edit</button>
</div>
```

button-clear 类可以设置无背景和边框的头部/底部按钮。

```
<div class="bar bar-header">
  <button class="button button-icon icon ion-navicon"></button>
  <div class="h1 title">Header Buttons</div>
  <button class="button button-clear button-positive">Edit</button>
</div>
```

按钮栏

我们可以使用 button-bar 类来设置按钮栏。以下实例中，我们在头部和内容中添加了按钮栏。

```
<div class="button-bar">
  <a class="button">First</a>
  <a class="button">Second</a>
  <a class="button">Third</a>
</div>
```

ionic 列表

列表是一个应用广泛的界面元素，在所有移动app中几乎都会使用到。

列表可以是基本文字、按钮，开关，图标和缩略图等。

列表项可以是任何的HTML元素。容器元素需要list类，每个列表项需要使用item类。

ionList和ionItem可以很容易的支持各种交互方式，比如，滑动编辑，拖动排序，以及删除项。

基本用法:

```
<ul class="list">
    <li class="item">
      ...
    </li>
</ul>
```

列表分隔符

我们可以使用 item-divider 类来为列表创建分隔符，默认情况下，列表项以不同的背景颜色和字体加粗来区分，但你也可以很容易的定制他。

```
<div class="list">

  <div class="item item-divider">
    Candy Bars
  </div>

  <a class="item" href="#">
    Butterfinger
  </a>

  ...

</div>
```

带图标列表

我们可以在列表项的左侧或右侧指定图标。

使用 item-icon-left 图标在左侧， item-icon-right 设置图标在右侧。如果你需要在两边都有图标，则两个类都添加上即可。

以下实例中，我们在列表项中使用了 <a> 标签，使得每个列表项可点击。

列表项在使用 <a>或<button> 元素时，如果右侧未添加图标，则会自动添加上箭头号。

实例中，第一项只有左侧图标，第二项左右均有图标，第三项有右侧图标（还有注释 item-note），第四项有badge（标记）元素。

```
<div class="list">

  <a class="item item-icon-left" href="#">
    <i class="icon ion-email"></i>
    Check mail
  </a>

  <a class="item item-icon-left item-icon-right" href="#">
    <i class="icon ion-chatbubble-working"></i>
    Call Ma
    <i class="icon ion-ios-telephone-outline"></i>
  </a>

  <a class="item item-icon-left" href="#">
    <i class="icon ion-mic-a"></i>
    Record album
    <span class="item-note">
      Grammy
    </span>
  </a>

  <a class="item item-icon-left" href="#">
    <i class="icon ion-person-stalker"></i>
    Friends
    <span class="badge badge-assertive">0</span>
  </a>

</div>
```

按钮列表

使用 item-button-right 或 item-button-left 类将按钮放在列表项中。

```
<div class="list">

  <div class="item item-button-right">
    Call Ma
    <button class="button button-positive">
      <i class="icon ion-ios-telephone"></i>
    </button>
  </div>

  ...

</div>
```

带头像列表

使用 item-avatar 来创建一个带头像的列表：

```
<div class="list">

    <a class="item item-avatar" href="#">
      <img src="venkman.jpg">
      <h2>Venkman</h2>
      <p>Back off, man. I'm a scientist.</p>
    </a>

    ...

</div>
```

缩略图列表

item-thumbnail-left 类用于添加左侧对齐的缩略图， item-thumbnail-right 类用于添加右侧对齐的缩略图。

```
<div class="list">

    <a class="item item-thumbnail-left" href="#">
      <img src="cover.jpg">
      <h2>Pretty Hate Machine</h2>
      <p>Nine Inch Nails</p>
    </a>

    ...

</div>
```

内嵌列表(inset list)

我们可以在容器当中内嵌列表，列表不会显示完整的宽度。

内嵌列表的样式为：list list-inset，与常规列表区别是，它设置了外边距（marign）,类似于选项卡。

内嵌列表是没有阴影效果的，滚动时效果会更好。

```
<div class="list list-inset">

    <div class="item">
      Raiders of the Lost Ark
    </div>

    ...

</div>
```

ionic 卡片

近年来卡片(card)的应用越来越流行，卡片提供了一个更好组织信息展示的工具。

针对移动端的应用，卡片会根据屏幕大小自适应大小。

我们可以很灵活的控制卡片的显示效果，甚至实现动画效果。

卡片一般放在页面顶部，当然也可以实现左右切换的功能。

```
<div class="card">
  <div class="item item-text-wrap">
    基本卡片，包含了文本信息。
  </div>
</div>
```

卡片(card)默认样式带有box-shadow(阴影)，由于性能的原因，和他类似的元素像 list list-inset 并没有阴影。

如果你有很多的卡片，每个卡片都有很多子元素，建议使用内嵌列表（inset list）。

卡片的头部与底部

我们可以通过添加 item-divider 类为卡片添加头部与底部：

```
<div class="card">
  <div class="item item-divider">
    卡片头部！
  </div>
  <div class="item item-text-wrap">
    基本卡片，包含了文本信息。
  </div>
  <div class="item item-divider">
    卡片底部！
  </div>
</div>
```

卡片列表

使用 list card 类来设置卡片列表：

```
<div class="list card">

  <a href="#" class="item item-icon-left">
    <i class="icon ion-home"></i>
    Enter home address
  </a>

  <a href="#" class="item item-icon-left">
    <i class="icon ion-ios-telephone"></i>
    Enter phone number
  </a>

  <a href="#" class="item item-icon-left">
    <i class="icon ion-wifi"></i>
    Enter wireless password
  </a>

  <a href="#" class="item item-icon-left">
    <i class="icon ion-card"></i>
    Enter card information
  </a>

</div>
```

带图片卡片

卡片中使用图片，效果会更好，实例如下：

```
<div class="list card">

  <div class="item item-avatar">
    <img src="avatar.jpg">
    <h2>Pretty Hate Machine</h2>
    <p>Nine Inch Nails</p>
  </div>

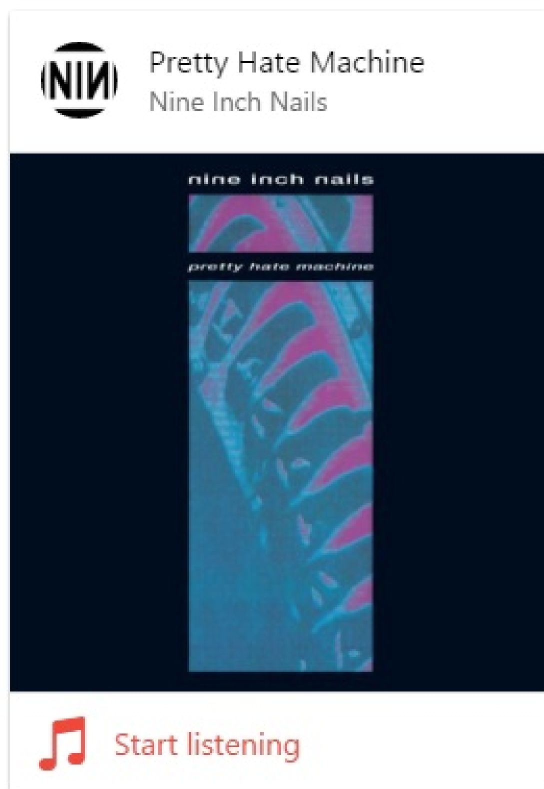
  <div class="item item-image">
    <img src="cover.jpg">
  </div>

  <a class="item item-icon-left assertive" href="#">
    <i class="icon ion-music-note"></i>
    Start listening
  </a>

</div>
```

运行效果如下：

卡片



卡片展现

以下实例中使用几种不同的选项的卡片展现方式。 开始使用了 list card 元素，并使用了 item-avatar , item-body 元素用于展示图片和文本信息，底部使用 item-divider 类。

```
<div class="list card">

  <div class="item item-avatar">
    <img src="mcfly.jpg">
    <h2>Marty McFly</h2>
    <p>November 05, 1955</p>
  </div>

  <div class="item item-body">
    <img class="full-image" src="delorean.jpg">
    <p>
      菜鸟教程 -- 学的不仅是技术，更新梦想！<br>
      菜鸟教程 -- 学的不仅是技术，更新梦想！<br>
      菜鸟教程 -- 学的不仅是技术，更新梦想！<br>
      菜鸟教程 -- 学的不仅是技术，更新梦想！
    </p>
    <p>
      <a href="#" class="subdued">1 喜欢</a>
      <a href="#" class="subdued">5 评论</a>
    </p>
  </div>

  <div class="item tabs tabs-secondary tabs-icon-left">
    <a class="tab-item" href="#">
      <i class="icon ion-thumbsup"></i>
      喜欢
    </a>
    <a class="tab-item" href="#">
      <i class="icon ion-chatbox"></i>
      Comment
    </a>
    <a class="tab-item" href="#">
      <i class="icon ion-share"></i>
      分享
    </a>
  </div>

</div>
```

运行效果如下：

卡片

Marty McFly
November 05, 1955

菜鸟教程 -- 学的不仅是技术，更新梦想！
菜鸟教程 -- 学的不仅是技术，更新梦想！
菜鸟教程 -- 学的不仅是技术，更新梦想！
菜鸟教程 -- 学的不仅是技术，更新梦想！

1 喜欢 5 评论

喜欢 评论 分享

ionic 表单和输入框

list 类同样可以用于 input 元素。item-input 和 item 类指定了文本框及其标签。

输入框属性：placeholder

以下实例中，默认为100%宽度（左右两侧没有边框），并使用 placeholder 属性设置输入字段预期值的提示信息。

```
<div class="list">
  <label class="item item-input">
    <input type="text" placeholder="First Name">
  </label>
  <label class="item item-input">
    <input type="text" placeholder="Last Name">
  </label>
  <label class="item item-input">
    <textarea placeholder="Comments"></textarea>
  </label>
</div>
```

输入框属性：input-label

使用 input-label 将标签放置于输入框 input 的左侧。

```
<div class="list">
  <label class="item item-input">
    <span class="input-label">用户名：</span>
    <input type="text">
  </label>
  <label class="item item-input">
    <span class="input-label">密码：</span>
    <input type="password">
  </label>
</div>
```

堆叠标签

堆叠标签通常位于输入框的头部。每个选项使用 item-stacked-label 类指定。 每个输入框需要指定 input-label。以下实例也使用了 placeholder 属性来设置信息输入提示。

```
<div class="list">
  <label class="item item-input item-stacked-label">
    <span class="input-label">First Name</span>
    <input type="text" placeholder="John">
  </label>
  <label class="item item-input item-stacked-label">
    <span class="input-label">Last Name</span>
    <input type="text" placeholder="Suhr">
  </label>
  <label class="item item-input item-stacked-label">
    <span class="input-label">Email</span>
    <input type="text" placeholder="john@suhr.com">
  </label>
</div>
```

浮动标签

浮动标签类似于堆叠标签，但浮动标签有一个动画的效果，每个选项需要指定 item-floating-label 类，输入标签需要指定 input-label。

```
<div class="list">
  <label class="item item-input item-floating-label">
    <span class="input-label">First Name</span>
    <input type="text" placeholder="First Name">
  </label>
  <label class="item item-input item-floating-label">
    <span class="input-label">Last Name</span>
    <input type="text" placeholder="Last Name">
  </label>
  <label class="item item-input item-floating-label">
    <span class="input-label">Email</span>
    <input type="text" placeholder="Email">
  </label>
</div>
```

内嵌表单

默认情况下每个输入域宽度都是100%，但我们可以使用 list list-inset 或 card 类设置表单的内边距(padding)， card 类带有阴影。

```
<div class="list list-inset">
  <label class="item item-input">
    <input type="text" placeholder="First Name">
  </label>
  <label class="item item-input">
    <input type="text" placeholder="Last Name">
  </label>
</div>
```

内嵌输入域

使用 list-inset 设置内嵌实体列表。 使用 item-input-inset 样式可以内嵌一个按钮。

```
<div class="list">

  <div class="item item-input-inset">
    <label class="item-input-wrapper">
      <input type="text" placeholder="Email">
    </label>
    <button class="button button-small">
      Submit
    </button>
  </div>

</div>
```

带图标的输入框

item-input 输入框可以很简单的添加图标。 图标可以在 <input> 前添加。

```
<div class="list list-inset">
  <label class="item item-input">
    <i class="icon ion-search placeholder-icon"></i>
    <input type="text" placeholder="Search">
  </label>
</div>
```

头部输入框

输入框可放置在头部，并可添加提交或取消按钮。

```
<div class="bar bar-header item-input-inset">
  <label class="item-input-wrapper">
    <i class="icon ion-ios-search placeholder-icon"></i>
    <input type="search" placeholder="搜索">
  </label>
  <button class="button button-clear">
    取消
  </button>
</div>
```

ionic Toggle(切换开关)

切换开关类似与 HTML 的 checkbox 标签，但它更易于在移动设备上使用。

切换开关可以使用 toggle-assertive 来指定颜色。

```
<label class="toggle">
   <input type="checkbox">
   <div class="track">
     <div class="handle"></div>
   </div>
</label>
```

该实例有是多个切换开关列表。注意，每个选项的 item 类后需要添加 item-toggle 类。

```
<ul class="list">

  <li class="item item-toggle">
     HTML5
     <label class="toggle toggle-assertive">
       <input type="checkbox">
       <div class="track">
         <div class="handle"></div>
       </div>
     </label>
  </li>

  ...

</ul>
```

运行效果如下：

切换

HTML5

CSS3

Flashplayer

Java Applets

JavaScript

Silverlight

Web Components

ionic checkbox（复选框）

ionic 里面的 Checkbox 和普通的 Checkbox 效果上其实相差不大，只是样式上有所不同。

以下实例颜色了多个复选框的列表。

注意，每个选项的 item 类后需要添加 item-checkbox 类。

复选框可以使用 checkbox-assertive 来指定颜色。

```
<ul class="list">

  <li class="item item-checkbox">
     <label class="checkbox">
       <input type="checkbox">
     </label>
     Flux Capacitor
  </li>

  ...

</ul>
```

运行效果如下：

复选框

- [x] Flux Capacitor
- [x] 1.21 Gigawatts
- [x] Delorean
- [x] 88 MPH
- [ ] Plutonium Resupply

ionic 单选框

ionic 当选按钮与标准 type="radio" 的 input元素类似。以下展示了现代app类型的单选按钮。

每个 item-radio 中的 type="radio" 的 input 元素的 name 属性都相同。radio-icon 类用于是否显示图标。

ionic 在单选项中使用了 <label> 元素，使其更易点击。

实例

```
<div class="list">

<label class="item item-radio">
  <input type="radio" name="group" value="go" checked="checked">
  <div class="item-content">
    Go
  </div>
  <i class="radio-icon ion-checkmark"></i>
</label>

<label class="item item-radio">
  <input type="radio" name="group" value="python">
  <div class="item-content">
    Python
  </div>
  <i class="radio-icon ion-checkmark"></i>
</label>

<label class="item item-radio">
  <input type="radio" name="group" value="ruby">
  <div class="item-content">
    Ruby
  </div>
  <i class="radio-icon ion-checkmark"></i>
</label>

<label class="item item-radio">
  <input type="radio" name="group" value=".net">
  <div class="item-content">
    .Net
  </div>
  <i class="radio-icon ion-checkmark"></i>
</label>

<label class="item item-radio">
  <input type="radio" name="group" value="java">
```

```
  <div class="item-content">
    Java
  </div>
  <i class="radio-icon ion-checkmark"></i>
</label>

<label class="item item-radio">
  <input type="radio" name="group" value="php">
  <div class="item-content">
    PHP
  </div>
  <i class="radio-icon ion-checkmark"></i>
</label>

</div>
```

运行效果如下：

单选按钮列表

Go ✔

Python

Ruby

.Net

Java

PHP

ionic Range

ionic Range 是一个滑块控件，ionic 为 Range 提供了很多种默认的样式。而且你可以在许多种元素里使用它比如列表或者 Card 。

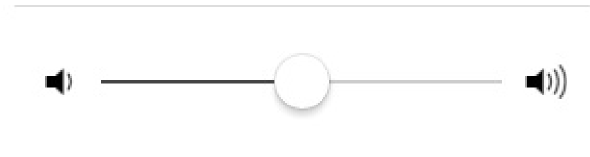
实例

```
<div class="range">
  <i class="icon ion-volume-low"></i>
  <input type="range" name="volume">
  <i class="icon ion-volume-high"></i>
</div>

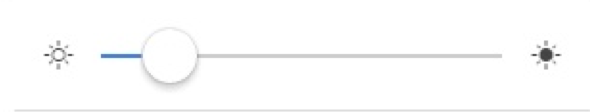
<div class="list" style="margin-top: 13px">
  <div class="item item-divider">
    Ranges In A List
  </div>
  <div class="item range range-positive">
    <i class="icon ion-ios-sunny-outline"></i>
    <input type="range" name="volume" min="0" max="100" value="12">
    <i class="icon ion-ios-sunny"></i>
  </div>
  <div class="item range range-calm">
    <i class="icon ion-ios-lightbulb-outline"></i>
    <input type="range" name="volume" min="0" max="100" value="25">
    <i class="icon ion-ios-lightbulb"></i>
  </div>
  <div class="item range range-balanced">
    <i class="icon ion-ios-bolt-outline"></i>
    <input type="range" name="volume" min="0" max="100" value="38">
    <i class="icon ion-ios-bolt"></i>
  </div>
  <div class="item range range-energized">
    <i class="icon ion-ios-moon-outline"></i>
    <input type="range" name="volume" min="0" max="100" value="50">
    <i class="icon ion-ios-moon"></i>
  </div>
  <div class="item range range-assertive">
    <i class="icon ion-ios-partlysunny-outline"></i>
    <input type="range" name="volume" min="0" max="100" value="63">
    <i class="icon ion-ios-partlysunny"></i>
  </div>
  <div class="item range range-royal">
    <i class="icon ion-ios-rainy-outline"></i>
    <input type="range" name="volume" min="0" max="100" value="75">
    <i class="icon ion-ios-rainy"></i>
  </div>
  <div class="item range range-dark">
    <i class="icon ion-ios-lightbulb-outline"></i>
    <input type="range" name="volume" min="0" max="100" value="88">
    <i class="icon ion-ios-lightbulb"></i>
  </div>
</div>
```

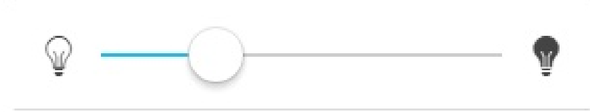
运行效果如下：

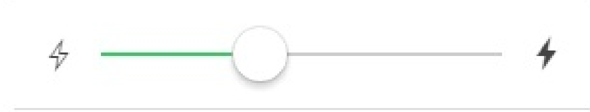
Range（滑块控件）

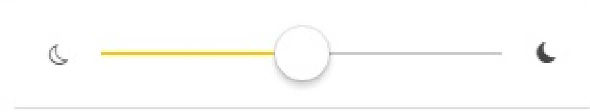


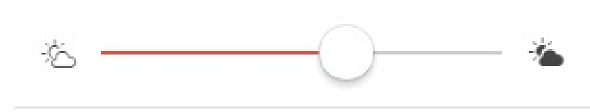
Ranges In A List

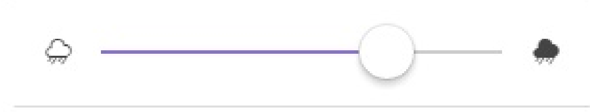


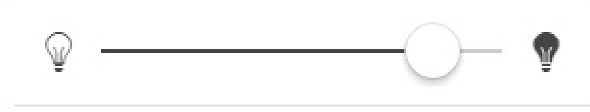












ionic select

ionic select 的 select 相比原生的要更加美观些。但是弹出的可选选项样式是浏览器默认的。

每个平台上的可选项样式都是不一样的，在PC电脑的浏览器上，你会看到传统的下拉界面，Android 上会弹出单选按钮选项，iOS 有个滚轮操作界面。

实例

```
<div  class="list">  <div  class="item item-input item-select">  <c
```

运行效果如下：

Select

Lightsaber	Green
Fighter	X-wing
Droid	R2-D2
Planet	Dagobah

ionic tab(选项卡)

ionic tab(选项卡) 是水平排列的按钮或者链接，用以页面间导航的切换。它可以包含文字和图标的组合，是一种移动设备上流行的导航方法。

以下选项卡容器使用了 tabs 类，每个选项卡使用 tab-item 类。默认情况下，选项卡是文本的，并没有图标。

实例

```
<div class="tabs">
  <a class="tab-item">
    主页
  </a>
  <a class="tab-item">
    收藏
  </a>
  <a class="tab-item">
    设置
  </a>
</div>
```

默认情况，选项卡颜色为默认，你可以设置以下不同颜色样式：tabs-default，tabs-light， tabs-stable， tabs-positive， tabs-calm， tabs-balanced， tabs-energized， tabs-assertive， tabs-royal， tabs-dark。

要隐藏选项卡栏，可使用 tabs-item-hide 类。

图标选项卡

在 tabs 类后添加 tabs-icon-only 类可设置只显示图标选项卡。

```
<div class="tabs tabs-icon-only">
  <a class="tab-item">
    <i class="icon ion-home"></i>
  </a>
  <a class="tab-item">
    <i class="icon ion-star"></i>
  </a>
  <a class="tab-item">
    <i class="icon ion-gear-a"></i>
  </a>
</div>
```

顶部图标+文字选项卡

在 tabs 类后添加 ttabs-icon-top 类可设置顶部图标+文字选项卡。

```
<div class="tabs tabs-icon-top">
  <a class="tab-item" href="#">
    <i class="icon ion-home"></i>
    主页
  </a>
  <a class="tab-item" href="#">
    <i class="icon ion-star"></i>
    收藏
  </a>
  <a class="tab-item" href="#">
    <i class="icon ion-gear-a"></i>
    设置
  </a>
</div>
```

左侧图标+文字选项卡

在 tabs 类后添加 ttabs-icon-left 类可设置左侧图标+文字选项卡。

```
<div class="tabs tabs-icon-left">
  <a class="tab-item">
    <i class="icon ion-home"></i>
    主页
  </a>
  <a class="tab-item">
    <i class="icon ion-star"></i>
    收藏
  </a>
  <a class="tab-item">
    <i class="icon ion-gear-a"></i>
    设置
  </a>
</div>
```

条纹样式选项卡

可以在带有 tabs 的样式名的元素上添加 tabs-striped 来实现像 Android 风格的 tabs。也可以添加 tabs-top 来实现选项卡在页面顶部。

条纹选项卡颜色可通过 tabs-background-{color} 和 tabs-color-{color} 来控制，{color} 值可以是：default, light, stable, positive, calm, balanced, energized, assertive, royal, 或 dark。

注意：如果要再选项卡上设置头部标题，需要使用 has-tabs-top 类。

```
<div class="tabs-striped tabs-top tabs-background-positive tabs-col
    <div class="tabs">
      <a class="tab-item active" href="#">
        <i class="icon ion-home"></i>
        Test
      </a>
      <a class="tab-item" href="#">
        <i class="icon ion-star"></i>
        Favorites
      </a>
      <a class="tab-item" href="#">
        <i class="icon ion-gear-a"></i>
        Settings
      </a>
    </div>
  </div>
  <div class="tabs-striped tabs-color-assertive">
    <div class="tabs">
      <a class="tab-item active" href="#">
        <i class="icon ion-home"></i>
        Test
      </a>
      <a class="tab-item" href="#">
        <i class="icon ion-star"></i>
        Favorites
      </a>
      <a class="tab-item" href="#">
        <i class="icon ion-gear-a"></i>
        Settings
      </a>
    </div>
  </div>
```

运行效果如下：


条纹样式选项卡



ionic 网格(Grid)

ionic 的网格(Grid)和其他大部分框架有所不同，它采用了弹性盒子模型(Flexible Box Model) 。而且在移动端，基本上的手机都支持。row 样式指定行，col 样式指定列。

同等大小网格

在带有 row 的样式的元素里如果包含了 col 的样式，col 就会设置为同等大小。

以下实例中 row 的样式包含了 5 个 col 样式，每个 col 的宽度为 20%。

```
<div class="row">
  <div class="col">.col</div>
  <div class="col">.col</div>
  <div class="col">.col</div>
  <div class="col">.col</div>
  <div class="col">.col</div>
</div>
```

指定列宽

你可以设定一行中各个列的大小不一样。默认情况下，列都会被划分为同等大小。但你也可以按百分比来设置列的宽度（一行为 12 个网格）。

```
<div class="row">
  <div class="col col-50">.col.col-50</div>
  <div class="col">.col</div>
  <div class="col">.col</div>
</div>

<div class="row">
  <div class="col col-75">.col.col-75</div>
  <div class="col">.col</div>
</div>

<div class="row">
  <div class="col">.col</div>
  <div class="col col-75">.col.col-75</div>
</div>

<div class="row">
  <div class="col">.col</div>
  <div class="col">.col</div>
</div>
```

注意：实例中，每个 col 样式会自动添加上边框和灰色背景。

下面列出了指定列宽的一些百分比的样式名:

.col-10	10%
.col-20	20%
.col-25	25%
.col-33	33.3333%
.col-50	50%
.col-67	66.6666%
.col-75	75%
.col-80	80%
.col-90	90%

有偏移量的网格

列可以设置左侧偏移量，实例如下：

```
<div class="row">
  <div class="col col-33 col-offset-33">.col</div>
  <div class="col">.col</div>
</div>

<div class="row">
  <div class="col col-33">.col</div>
  <div class="col col-33 col-offset-33">.col</div>
</div>

<div class="row">
  <div class="col col-33 col-offset-67">.col</div>
</div>
```

下面是一些百分比的偏移量样式名：

| .col-offset-10 | 10% | | .col-offset-20 | 20% | | .col-offset-25 | 25% | | .col-offset-33 | 33.3333% | | .col-offset-50 | 50% | | .col-offset-67 | 66.6666% | | .col-offset-75 | 75% | | .col-offset-80 | 80% | | .col-offset-90 | 90% |

纵向对齐网格

弹性盒子模型可以很容易设置列纵向对齐。纵向对齐包含顶部，中间部分，底部，可以应用到每一行的列，或者指定的某列。

实例中，最后一列设置了最高的内容用于更好的演示纵向对齐网格。

```
<div class="row">
  <div class="col">.col</div>
  <div class="col">.col</div>
  <div class="col">.col</div>
  <div class="col">1<br>2<br>3<br>4</div>
</div>

<div class="row">
  <div class="col col-top">.col</div>
  <div class="col col-center">.col</div>
  <div class="col col-bottom">.col</div>
  <div class="col">1<br>2<br>3<br>4</div>
</div>

<div class="row row-top">
  <div class="col">.col</div>
  <div class="col">.col</div>
  <div class="col">.col</div>
  <div class="col">1<br>2<br>3<br>4</div>
</div>

<div class="row row-center">
  <div class="col">.col</div>
  <div class="col">.col</div>
  <div class="col">.col</div>
  <div class="col">1<br>2<br>3<br>4</div>
</div>

<div class="row row-bottom">
  <div class="col">.col</div>
  <div class="col">.col</div>
  <div class="col">.col</div>
  <div class="col">1<br>2<br>3<br>4</div>
</div>
```

响应式网格

手持设备屏幕在切换时，例如横屏，竖屏等。就需要设置每行的网格可以实现根据不同宽度自适应大小。

不同设备响应式类的样式如下:

| 响应式类 | 描述 | | .responsive-sm | 小于手机横屏 | | .responsive-md | 小于平板竖屏 | | .responsive-lg | 小于平板横屏 |

```
<div class="row responsive-sm">
  <div class="col">.col</div>
  <div class="col">.col</div>
  <div class="col">.col</div>
  <div class="col">.col</div>
</div>
```

ionic 颜色

ionic 提供了很多颜色的配置，当然你可以根据自己的需要自定义颜色。

```
<ul class="list color-list-demo">
    <li class="item dark">
      light
      <span class="color-demo light-bg light-border"></span>
    </li>
    <li class="item stable-dark">
      stable
      <span class="color-demo stable-bg stable-border"></span>
    </li>
    <li class="item positive">
      positive
      <span class="color-demo positive-bg positive-border"></span>
    </li>
    <li class="item calm">
      calm
      <span class="color-demo calm-bg calm-border"></span>
    </li>
    <li class="item balanced">
      balanced
      <span class="color-demo balanced-bg balanced-border"></span>
    </li>
    <li class="item energized">
      energized
      <span class="color-demo energized-bg energized-border"></spa
    </li>
    <li class="item assertive">
      assertive
      <span class="color-demo assertive-bg assertive-border"></spa
    </li>
    <li class="item royal">
      royal
      <span class="color-demo royal-bg royal-border"></span>
    </li>
    <li class="item dark">
      dark
      <span class="color-demo dark-bg dark-border"></span>
    </li>
</ul>
```

实例运行结果：

颜色列表

light

stable

positive

calm

balanced

energized

assertive

royal

dark

ionic icon(图标)

ionic 也默认提供了许多的图标，大概有500多个。用法也非常的简单：

```
<i  class="icon icon ion-star"></i>
```

图标样式CDN地址：**http://www.runoob.com/static/ionic/css/ionicons.min.css**。

图标列表如下：

-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-

ionic JavaScript

ionic 上拉菜单(ActionSheet)

上拉菜单(ActionSheet)通过往上弹出的框，来让用户选择选项。

非常危险的选项会以高亮的红色来让人第一时间识别。你可以通过点击取消按钮或者点击空白的地方来让它消失。

实例

HTML 代码

```
<body ng-app="starter" ng-controller="actionsheetCtl" >

    <ion-pane>
        <ion-content >
            <h2 ng-click="show()">Action Sheet</h2>
        </ion-content>
    </ion-pane>
</body>
```

JavaScript 代码

在代码中触发上拉菜单，需要在你的 angular 控制器中使用 $ionicActionSheet 服务：

```
angular.module('starter', ['ionic'])

.run(function($ionicPlatform) {
  $ionicPlatform.ready(function() {
    // Hide the accessory bar by default (remove this to show the a
    // for form inputs)
    if(window.cordova && window.cordova.plugins.Keyboard) {
      cordova.plugins.Keyboard.hideKeyboardAccessoryBar(true);
    }
    if(window.StatusBar) {
      StatusBar.styleDefault();
    }
  });
})

.controller( 'actionsheetCtl',['$scope','$ionicActionSheet','$timeo
    $scope.show = function() {

        var hideSheet = $ionicActionSheet.show({
            buttons: [
              { text: '<b>Share</b> This' },
              { text: 'Move' }
            ],
            destructiveText: 'Delete',
            titleText: 'Modify your album',
            cancelText: 'Cancel',
            cancel: function() {
                 // add cancel code..
               },
            buttonClicked: function(index) {
              return true;
            }
        });

        $timeout(function() {
            hideSheet();
        }, 2000);

    };
}])
```

运行效果如下图：



ionic 背景层

我们经常需要在 UI 上，例如在弹出框、加载框、其他弹出层中显示或隐藏背景层。

在组件中可以使用$ionicBackdrop.retain()来显示背景层，使用$ionicBackdrop.release()隐藏背景层。

每次调用retain后，背景会一直显示，直到调用release消除背景层。

实例

HTML 代码

```
<body ng-app="starter" ng-controller="actionsheetCtl" >
    <ion-pane>
        <ion-content >
            <h2 ng-click="action()">$ionicBackdrop</h2>
        </ion-content>
    </ion-pane>
</body>
```

JavaScript 代码

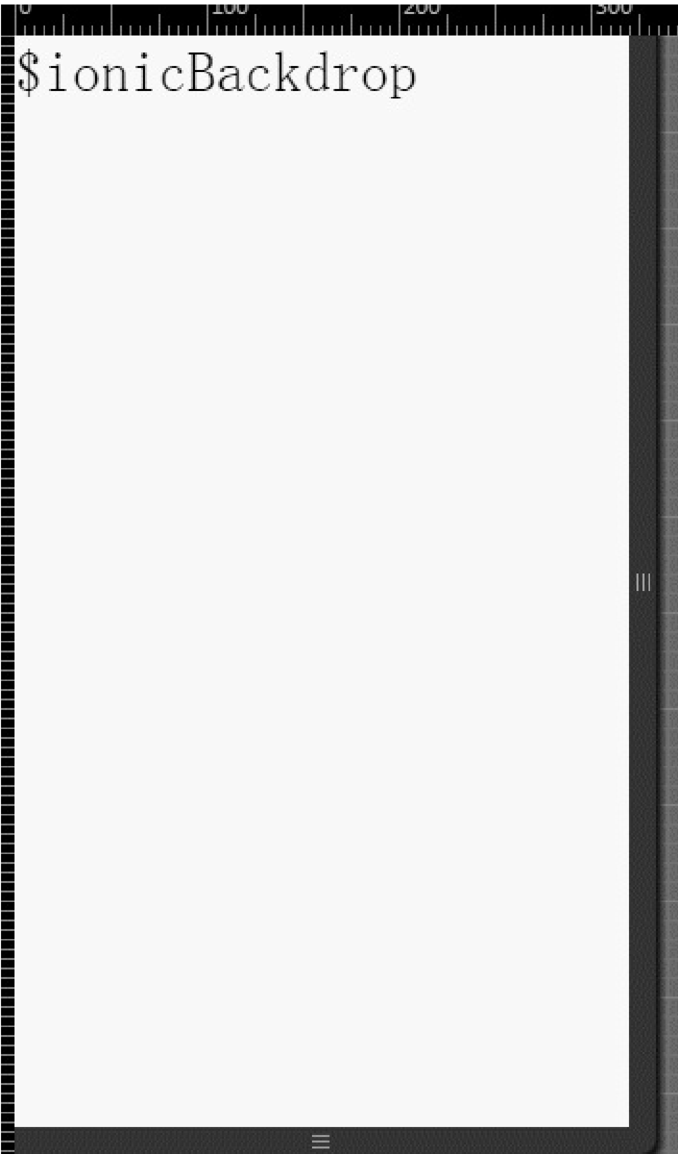
```
angular.module('starter', ['ionic'])

.run(function($ionicPlatform) {
  $ionicPlatform.ready(function() {
    // Hide the accessory bar by default (remove this to show the a
    // for form inputs)
    if(window.cordova && window.cordova.plugins.Keyboard) {
      cordova.plugins.Keyboard.hideKeyboardAccessoryBar(true);
    }
    if(window.StatusBar) {
      StatusBar.styleDefault();
    }
  });
})

.controller( 'actionsheetCtl',['$scope','$timeout' ,'$ionicBackdrop

    $scope.action = function() {
       $ionicBackdrop.retain();
       $timeout(function() {    //默认让它1秒后消失
         $ionicBackdrop.release();
       }, 1000);
    };
}])
```

显示效果如下图所示：


$ionicBackdrop

ionic 下拉刷新

在加载新数据的时候，我们需要实现下拉刷新效果，代码如下：

实例

HTML 代码

```
<body ng-app="starter" ng-controller="actionsheetCtl" >
    <ion-pane>
        <ion-content >
            <ion-refresher pulling-text="下拉刷新" on-refresh="doRef
            <ion-list>
                <ion-item ng-repeat="item in items" ng-bind="item.n
            </ion-list>
        </ion-content>
    </ion-pane>
</body>
```

JavaScript 代码

```
angular.module('starter', ['ionic'])

.run(function($ionicPlatform) {
  $ionicPlatform.ready(function() {
    // Hide the accessory bar by default (remove this to show the a
    // for form inputs)
    if(window.cordova && window.cordova.plugins.Keyboard) {
      cordova.plugins.Keyboard.hideKeyboardAccessoryBar(true);
    }
    if(window.StatusBar) {
      StatusBar.styleDefault();
    }
  });
})

.controller( 'actionsheetCtl',['$scope','$timeout' ,'$http',functio

    $scope.items=[
        {
            "name":"HTML5"
        },
        {
            "name":"JavaScript"
        },
        {
            "name":"Css3"
        }
    ];

    $scope.doRefresh = function() {
        $http.get('http://www.runoob.com/try/demo_source/item.json
            .success(function(newItems) {
                $scope.items = newItems;
            })
            .finally(function() {
                $scope.$broadcast('scroll.refreshComplete');
            });
    };
}])
```

item.json 文件数据：

```
[
    {
        "name":"菜鸟教程"
    },
    {
        "name":"www.runoob.com"
    }
]
```

效果如下所示：

HTML5

JavaScript

Css3

ionic 复选框

ionic 复选框（checkbox）与普通的 HTML 复选框没什么区别，以下实例演示了 ionic 复选框 ion-checkbox 的应用。

```
<ion-checkbox ng-model="isChecked">复选框标签</ion-checkbox>
```

实例

实例中，会根据复选框是否选中，修改 checked 值，true 为选中， false 为未选中。

HTML 代码

```
<ion-header-bar class="bar-positive">
  <h1 class="title">复选框</h1>
</ion-header-bar>

<ion-content>

  <div class="list">

    <ion-checkbox ng-repeat="item in devList"
                  ng-model="item.checked"
                  ng-checked="item.checked">
      {{ item.text }}
    </ion-checkbox>

    <div class="item">
      <div ng-bind="devList | json"></div>
    </div>

    <div class="item item-divider">
      Notifications
    </div>

    <ion-checkbox ng-model="pushNotification.checked"
                  ng-change="pushNotificationChange()">
      Push Notifications
    </ion-checkbox>

    <div class="item">
      <div ng-bind="pushNotification | json"></div>
    </div>

    <ion-checkbox ng-model="emailNotification"
                  ng-true-value="'Subscribed'"
                  ng-false-value="'Unubscribed'">
      Newsletter
    </ion-checkbox>
    <div class="item">
      <div ng-bind="emailNotification | json"></div>
    </div>

  </div>

</ion-content>
```

JavaScript 代码

```
angular.module('starter', ['ionic'])

.run(function($ionicPlatform) {
$ionicPlatform.ready(function() {
  // Hide the accessory bar by default (remove this to show the acc
  // for form inputs)
  if(window.cordova && window.cordova.plugins.Keyboard) {
    cordova.plugins.Keyboard.hideKeyboardAccessoryBar(true);
  }
  if(window.StatusBar) {
    StatusBar.styleDefault();
  }
});
})

.controller( 'actionsheetCtl',['$scope',function($scope){

  $scope.devList = [
      { text: "HTML5", checked: true },
      { text: "CSS3", checked: false },
      { text: "JavaScript", checked: false }
    ];

    $scope.pushNotificationChange = function() {
      console.log('Push Notification Change', $scope.pushNotificat:
    };

    $scope.pushNotification = { checked: true };
    $scope.emailNotification = 'Subscribed';

}])
```

css 代码：

```
body {
  cursor: url('http://www.runoob.com/try/demo_source/finger.png'),
}
```

效果如下所示：

复选框


HTML5


CSS3


JavaScript


```
[
  {
    "text": "HTML5",
    "checked": true
  },
  {
    "text": "CSS3",
    "checked": false
  }
```

ionic 单选框操作

实例中，根据选中的不同选项，显示不同的值。

HTML 代码

```
<ion-header-bar class="bar-positive">
  <h1 class="title">当选按钮</h1>
</ion-header-bar>

<ion-content>

  <div class="list">

    <div class="item item-divider">
      选取的值为: {{ data.clientSide }}
    </div>

    <ion-radio ng-repeat="item in clientSideList"
               ng-value="item.value"
               ng-model="data.clientSide">
      {{ item.text }}
    </ion-radio>

    <div class="item item-divider">
      Serverside, Selected Value: {{ data.serverSide }}
    </div>

    <ion-radio ng-repeat="item in serverSideList"
               ng-value="item.value"
               ng-model="data.serverSide"
               ng-change="serverSideChange(item)"
               name="server-side">
      {{ item.text }}
    </ion-radio>

  </div>

</ion-content>
```

JavaScript 代码

```
angular.module('ionicApp', ['ionic'])

.controller('MainCtrl', function($scope) {

  $scope.clientSideList = [
    { text: "Backbone", value: "bb" },
    { text: "Angular", value: "ng" },
    { text: "Ember", value: "em" },
    { text: "Knockout", value: "ko" }
  ];

  $scope.serverSideList = [
    { text: "Go", value: "go" },
    { text: "Python", value: "py" },
    { text: "Ruby", value: "rb" },
    { text: "Java", value: "jv" }
  ];

  $scope.data = {
    clientSide: 'ng'
  };

  $scope.serverSideChange = function(item) {
    console.log("Selected Serverside, text:", item.text, "value:",
  };

});
```

css 代码：

```
body {
  cursor: url('http://www.runoob.com/try/demo_source/finger.png'),
}
```

效果如下所示：

当选按钮

Backbone

Angular ✔

Ember

Knockout

Serverside, Selected Value:

Go

Python

ionic 切换开关操作

以下实例中，通过切换不同开关 checked 显示不同的值，true 为打开，false 为关闭。

HTML 代码

```
<ion-header-bar class="bar-positive">
  <h1 class="title">开关切换</h1>
</ion-header-bar>

<ion-content>

  <div class="list">

    <div class="item item-divider">
      Settings
    </div>

    <ion-toggle ng-repeat="item in settingsList"
                ng-model="item.checked"
                ng-checked="item.checked">
      {{ item.text }}
    </ion-toggle>

    <div class="item">
        <!-- 使用 pre 标签展示效果更美观 -->
      <div ng-bind="settingsList | json"></div>
    </div>

    <div class="item item-divider">
      Notifications
    </div>

    <ion-toggle ng-model="pushNotification.checked"
                ng-change="pushNotificationChange()">
      Push Notifications
    </ion-toggle>

    <div class="item">
        <!-- 使用 pre 标签展示效果更美观 -->
      <div ng-bind="pushNotification | json"></div>
    </div>

    <ion-toggle toggle-class="toggle-assertive"
                ng-model="emailNotification"
                ng-true-value="Subscribed"
                ng-false-value="Unubscribed">
```

```
      Newsletter
    </ion-toggle>

    <div class="item">
        <!-- 使用 pre 标签展示效果更美观 -->
      <div ng-bind="emailNotification | json"></div>
    </div>

  </div>

</ion-content>
```

由于pre标签冲突，实例中的 pre 已替换为 div标签，具体可以在"尝试一下"中查看。

JavaScript 代码

```
angular.module('ionicApp', ['ionic'])

.controller('MainCtrl', function($scope) {

  $scope.settingsList = [
    { text: "Wireless", checked: true },
    { text: "GPS", checked: false },
    { text: "Bluetooth", checked: false }
  ];

  $scope.pushNotificationChange = function() {
    console.log('Push Notification Change', $scope.pushNotificatio
  };

  $scope.pushNotification = { checked: true };
  $scope.emailNotification = 'Subscribed';

});
```

css 代码：

```
body {
  cursor: url('http://www.runoob.com/try/demo_source/finger.png'),
}
```

效果如下所示：

切换开关

Settings

Wireless

GPS

Bluetooth

```
[
  {
    "text": "Wireless",
    "checked": true
  },
  {
    "text": "GPS",
```

ionic 手势事件

`on-hold` ：长按的时间是500毫秒。

```
&lt;button
    on-hold="onHold()"
    class="button"&gt;
    Test
    &lt;/button&gt;
```

`on-tap` ：这个是手势轻击事件，如果长按时间超过250毫秒，那就不是轻击了。。

```
&lt;button
    on-tap="onTap()"
    class="button"&gt;
    Test
    &lt;/button&gt;
```

`on-double-tap` ：手双击屏幕事件

```
&lt;button
    on-double-tap="onDoubleTap()"
    class="button"&gt;
    Test
    &lt;/button&gt;
```

`on-touch` ：这个和 on-tap 还是有区别的，这个是立即执行，而且是用户点击立马执行。不用等待 touchend/mouseup 。

```
    &lt;button on-touch="onTouch()"
    class="button"&gt;
    Test
    &lt;/button&gt;
```

`on-release` ：当用户结束触摸事件时触发。

```
&lt;button
    on-release="onRelease()"
    class="button"&gt;
    Test
&lt;/button&gt;
```

`on-drag` ：这个有点类似于PC端的拖拽。当你一直点击某个物体，并且手开始移动，都会触发 on-drag。

```
&lt;button
    on-drag="onDrag()"
    class="button"&gt;
    Test
&lt;/button&gt;
```

`on-drag-up` ：向上拖拽。

```
&lt;button
    on-drag-up="onDragUp()"
    class="button"&gt;
    Test
&lt;/button&gt;
```

`on-drag-right` ：向右拖拽。

```
&lt;button
    on-drag-right="onDragRight()"
    class="button"&gt;
    Test
&lt;/button&gt;
```

`on-drag-down` ：向下拖拽。

```
&lt;button
    on-drag-down="onDragDown()"
    class="button"&gt;
    Test
&lt;/button&gt;
```

`on-drag-left` ：向左边拖拽。

```
&lt;button
    on-drag-left="onDragLeft()"
    class="button"&gt;
    Test
&lt;/button&gt;
```

`on-swipe` ：指手指滑动效果，可以是任何方向上的。而且也和 on-drag 类似，都有四个方向上单独的事件。

```
&lt;button
    on-swipe="onSwipe()"
    class="button"&gt;
    Test
&lt;/button&gt;
```

`on-swipe-up` ：向上的手指滑动效果。

```
&lt;button
    on-swipe-up="onSwipeUp()"
    class="button"&gt;
    Test
&lt;/button&gt;
```

`on-swipe-right` ：向右的手指滑动效果。

```
&lt;button
    on-swipe-right="onSwipeRight()"
    class="button"&gt;
    Test
&lt;/button&gt;
```

`on-swipe-down` ：向下的手指滑动效果。

```
&lt;button
    on-swipe-down="onSwipeDown()"
    class="button"&gt;
    Test
&lt;/button&gt;
```

`on-swipe-left` ：向左的手指滑动效果。

```
&lt;button
    on-swipe-left="onSwipeLeft()"
    class="button"&gt;
    Test
&lt;/button&gt;
```

$ionicGesture

一个angular服务展示ionicionic.EventController手势。

方法

```
on(eventType, callback, $element)
```

在一个元素上添加一个事件监听器。

`eventType` : `string`

监听的手势事件。

`callback` : `function(e)`

当手势事件发生时触发的事件。

`$element` : `element`

angular元素监听的事件。

`options` : `object`

对象。

```
off(eventType, callback, $element)
```

在一个元素上移除一个手势事件监听器。

`eventType` : `string`

移除监听的手势事件。

`callback` : `function(e)`

移除监听器。

`$element` : `element`

被监听事件的angular元素。

ionic 头部和底部

ion-header-bar

这个是固定在屏幕顶部的一个头部标题栏。如果给它加上'bar-subheader' 这个样式，它就是副标题。

用法

```
<ion-header-bar align-title="left" class="bar-positive">
  <div class="buttons">
    <button class="button" ng-click="doSomething()">Left Button</bu
  </div>
  <h1 class="title">Title!</h1>
  <div class="buttons">
    <button class="button">Right Button</button>
  </div>
</ion-header-bar>
<ion-content>
  Some content!
</ion-content>
```

API

`align-title_(optional)_` : `string`

这个是对齐 title 的。如果没有设置，它将会按照手机的默认排版(Ios的默认是居中，Android默认是居左)。它的值可以是 'left','center','right'。

`no-tap-scroll_(optional)_` : `boolean`

这个是设置 header-bar 是否跟随着内容的滚动而滚动，就是是否固定在顶部。它的值是布尔值（true/false）。

ion-footer-bar

知道了 ion-header-bar， 理解ion-footer-bar就轻松多啦！只是 ion-footer-bar 是在屏幕的底部。

用法

```
<ion-content>
  Some content!
</ion-content>
<ion-footer-bar align-title="left" class="bar-assertive">
  <div class="buttons">
    <button class="button">Left Button</button>
  </div>
  <h1 class="title">Title!</h1>
  <div class="buttons" ng-click="doSomething()">
    <button class="button">Right Button</button>
  </div>
</ion-footer-bar>
```

API

与 ion-header-bar 不同的是，ion-footer-bar 只有 align-title 这个 API。

`align-title(optional)` : `string`

这个是对齐 title 的。如果没有设置，它将会按照手机的默认排版(Ios的默认是居中，Android默认是居左)。它的值可以是 'left','center','right'。

ionic 列表操作

列表是一个应用广泛在几乎所有移动app中的界面元素。ionList 和 ionItem 这两个指令还支持多种多样的交互模式，比如移除其中的某一项，拖动重新排序，滑动编辑等等。

用法

```
<ion-list>
  <ion-item ng-repeat="item in items">
    Hello, {{item}}!
  </ion-item>
</ion-list>
```

高级用法: 缩略图，删除按钮，重新排序，滑动

```
<ion-list ng-controller="MyCtrl"
          show-delete="shouldShowDelete"
          show-reorder="shouldShowReorder"
          can-swipe="listCanSwipe">
  <ion-item ng-repeat="item in items"
            class="item-thumbnail-left">

    <img ng-src="{{item.img}}">
    <h2>{{item.title}}</h2>
    <p>{{item.description}}</p>
    <ion-option-button class="button-positive"
                       ng-click="share(item)">
      分享
    </ion-option-button>
    <ion-option-button class="button-info"
                       ng-click="edit(item)">
      编辑
    </ion-option-button>
    <ion-delete-button class="ion-minus-circled"
                       ng-click="items.splice($index, 1)">
    </ion-delete-button>
    <ion-reorder-button class="ion-navicon"
                        on-reorder="reorderItem(item, $fromIndex, $
    </ion-reorder-button>

  </ion-item>
</ion-list>
```

API

`delegate-handle(可选)` : `字符串`

该句柄定义带有 `$ionicListDelegate` 的列表。

`show-delete(可选)` : `布尔值`

列表项的删除按钮当前是显示还是隐藏。

`show-reorder(可选)` : `布尔值`

列表项的排序按钮当前是显示还是隐藏。

`can-swipe(可选)` : `布尔值`

列表项是否被允许滑动显示选项按钮。默认：true。

实例

HTML 代码：

```
<html ng-app="ionicApp">
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scal
    <title>Ionic List Directive</title>

    <link href="http://www.runoob.com/static/ionic/css/ionic.min.cs
    <script src="http://www.runoob.com/static/ionic/js/ionic.bundle
  </head>

  <body ng-controller="MyCtrl">

    <ion-header-bar class="bar-positive">
      <div class="buttons">
        <button class="button button-icon icon ion-ios-minus-outlin
          ng-click="data.showDelete = !data.showDelete; data.showRe
      </div>
      <h1 class="title">Ionic Delete/Option Buttons</h1>
      <div class="buttons">
        <button class="button" ng-click="data.showDelete = false; d
            Reorder
        </button>
      </div>
    </ion-header-bar>

    <ion-content>

      <!-- The list directive is great, but be sure to also checkou
```

```
      <ion-list show-delete="data.showDelete" show-reorder="data.sh

        <ion-item ng-repeat="item in items"
                  item="item"
                  href="#/item/{{item.id}}" class="item-remove-anir
          Item {{ item.id }}
          <ion-delete-button class="ion-minus-circled"
                             ng-click="onItemDelete(item)">
          </ion-delete-button>
          <ion-option-button class="button-assertive"
                             ng-click="edit(item)">
            Edit
          </ion-option-button>
          <ion-option-button class="button-calm"
                             ng-click="share(item)">
            Share
          </ion-option-button>
          <ion-reorder-button class="ion-navicon" on-reorder="moveI
        </ion-item>

      </ion-list>

    </ion-content>

  </body>
</html>
```

CSS 代码

```
body {
  cursor: url('http://www.runoob.com/try/demo_source/finger.png'),
}
```

JavaScript 代码

```
angular.module('ionicApp', ['ionic'])

.controller('MyCtrl', function($scope) {

  $scope.data = {
    showDelete: false
  };

  $scope.edit = function(item) {
    alert('Edit Item: ' + item.id);
```

```
    };
    $scope.share = function(item) {
      alert('Share Item: ' + item.id);
    };

    $scope.moveItem = function(item, fromIndex, toIndex) {
      $scope.items.splice(fromIndex, 1);
      $scope.items.splice(toIndex, 0, item);
    };

    $scope.onItemDelete = function(item) {
      $scope.items.splice($scope.items.indexOf(item), 1);
    };

    $scope.items = [
      { id: 0 },
      { id: 1 },
      { id: 2 },
      { id: 3 },
      { id: 4 },
      { id: 5 },
      { id: 6 },
      { id: 7 },
      { id: 8 },
      { id: 9 },
      { id: 10 },
      { id: 11 },
      { id: 12 },
      { id: 13 },
      { id: 14 },
      { id: 15 },
      { id: 16 },
      { id: 17 },
      { id: 18 },
      { id: 19 },
      { id: 20 },
      { id: 21 },
      { id: 22 },
      { id: 23 },
      { id: 24 },
      { id: 25 },
      { id: 26 },
      { id: 27 },
      { id: 28 },
      { id: 29 },
      { id: 30 },
      { id: 31 },
      { id: 32 },
      { id: 33 },
      { id: 34 },
      { id: 35 },
      { id: 36 },
      { id: 37 },
```

```
    { id: 38 },
    { id: 39 },
    { id: 40 },
    { id: 41 },
    { id: 42 },
    { id: 43 },
    { id: 44 },
    { id: 45 },
    { id: 46 },
    { id: 47 },
    { id: 48 },
    { id: 49 },
    { id: 50 }
  ];

});
```

ionic 加载动作

$ionicLoading 是 ionic 默认的一个加载交互效果。里面的内容也是可以在模板里面修改。

用法

```
angular.module('LoadingApp', ['ionic'])
.controller('LoadingCtrl', function($scope, $ionicLoading) {
  $scope.show = function() {
    $ionicLoading.show({
      template: 'Loading...'
    });
  };
  $scope.hide = function(){
    $ionicLoading.hide();
  };
});
```

方法

显示一个加载效果。

```
show(opts)
```

`opts` : `object`

loading指示器的选项。可用属性：

- `{string=}` `template` 指示器的html内容。
- `{string=}` `templateUrl` 一个加载html模板的url作为指示器的内容。
- `{boolean=}` `noBackdrop` 是否隐藏背景。默认情况下它会显示。
- `{number=}` `delay` 指示器延迟多少毫秒显示。默认为不延迟。
- `{number=}` `duration` 等待多少毫秒后自动隐藏指示器。默认情况下，指示器会一直显示，直到触发 `.hide()` 。

隐藏一个加载效果。

```
hide()
```

API

`delegate-handle(可选)` ： `字符串`

该句柄定义带有 `$ionicListDelegate` 的列表。

`show-delete(可选)` ： `布尔值`

列表项的删除按钮当前是显示还是隐藏。

`show-reorder(可选)` ： `布尔值`

列表项的排序按钮当前是显示还是隐藏。

`can-swipe(可选)` ： `布尔值`

列表项是否被允许滑动显示选项按钮。默认：true。

实例

HTML 代码：

```
<html ng-app="ionicApp">
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="initial-scale=1, maximum-scale=1

    <title>Ionic Modal</title>

     <link href="http://www.runoob.com/static/ionic/css/ionic.min.c
    <script src="http://www.runoob.com/static/ionic/js/ionic.bundle
  </head>
  <body ng-controller="AppCtrl">

      <ion-view title="Home">
        <ion-header-bar>
          <h1 class="title">The Stooges</h1>
        </ion-header-bar>
        <ion-content has-header="true">
          <ion-list>
            <ion-item ng-repeat="stooge in stooges" href="#">{{stoo
          </ion-list>
        </ion-content>
      </ion-view>

  </body>
</html>
```

JavaScript 代码

```
angular.module('ionicApp', ['ionic'])
.controller('AppCtrl', function($scope, $timeout, $ionicLoading) {

  // Setup the loader
  $ionicLoading.show({
    content: 'Loading',
    animation: 'fade-in',
    showBackdrop: true,
    maxWidth: 200,
    showDelay: 0
  });

  // Set a timeout to clear loader, however you would actually cal
  $timeout(function () {
    $ionicLoading.hide();
    $scope.stooges = [{name: 'Moe'}, {name: 'Larry'}, {name: 'Curly
  }, 2000);

});
```

$ionicLoadingConfig

设置加载的默认选项:

用法:

```
var app = angular.module('myApp', ['ionic'])
app.constant('$ionicLoadingConfig', {
  template: '默认加载模板……'
});
app.controller('AppCtrl', function($scope, $ionicLoading) {
  $scope.showLoading = function() {
    $ionicLoading.show(); //配置选项在 $ionicLoadingConfig 设置
  };
});
```

ionic 模型

$ionicModal

$ionicModal 可以遮住用户主界面的内容框。

你可以在你的 index 文件或者是其他文件内嵌入以下代码(里面的代码可以根据你自己的业务场景相应的改变)。

```
<script id="my-modal.html" type="text/ng-template">
  <ion-modal-view>
    <ion-header-bar>
      <h1 class="title">My Modal title</h1>
    </ion-header-bar>
    <ion-content>
      Hello!
    </ion-content>
  </ion-modal-view>
</script>
```

然后你就可以在你的 Controller 里面的注入 $ionicModal 。然后调用你刚刚写入的模板，进行初始化操作。就像下面的代码：

```
angular.module('testApp', ['ionic'])
.controller('MyController', function($scope, $ionicModal) {
  $ionicModal.fromTemplateUrl('my-modal.html', {
    scope: $scope,
    animation: 'slide-in-up'
  }).then(function(modal) {
    $scope.modal = modal;
  });
  $scope.openModal = function() {
    $scope.modal.show();
  };
  $scope.closeModal = function() {
    $scope.modal.hide();
  };
  //Cleanup the modal when we're done with it!
  $scope.$on('$destroy', function() {
    $scope.modal.remove();
  });
  // Execute action on hide modal
  $scope.$on('modal.hidden', function() {
    // Execute action
  });
  // Execute action on remove modal
  $scope.$on('modal.removed', function() {
    // Execute action
  });
});
```

方法

```
fromTemplate(templateString, options)
```

`templateString` : `字符串`

模板的字符串作为模型的内容。

`options` : `对象`

传递 ionicModal#initialize方法的选项。

返回: 对象, 一个ionicModal控制器的实例。

```
fromTemplateUrl(templateUrl, options)
```

`templateUrl` : `字符串`

载入模板的url。

`options` : `对象`

通过ionicModal#initialize方法传递对象。

返回： promise对象。Promises对象是CommonJS工作组提出的一种规范，目的是为异步编程提供统一接口。

ionicModal

由$ionicModal服务实例化。

提示：当你完成每个模块清除时，确保调用remove()方法，以避免内存泄漏。

注意：一个模块从它的初始范围广播出 'modal.shown' 和 'modal.hidden'，把自身作为一个参数来传递。

方法

```
initialize(可选)
```

创建一个新的模型控制器示例。

`options` : `对象`

一个选项对象具有一下属性：

- `{object=}` `范围` 子类的范围。默认：创建一个$rootScope子类。
- `{string=}` `动画` 带有显示或隐藏的动画。默认：'slide-in-up'
- `{boolean=}` `第一个输入框获取焦点` 当显示时，模型的第一个输入元素是否自动获取焦点。默认：false。
- `{boolean=}` backdropClickToClose` 点击背景时是否关闭模型。默认：true。

```
show()
```

显示模型实例

- 返回值: `promise` promise对象,在模型完成动画后得到解析

```
hide()
```

隐藏模型。

- 返回值: `promise` promise对象,在模型完成动画后得到解析

```
remove()
```

从 DOM 中移除模型实例并清理。

- 返回值: `promise` promise对象,在模型完成动画后得到解析

```
isShown()
```

- 返回：布尔值，用于判断模型是否显示。

实例

HTML 代码

```
<html ng-app="ionicApp">
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="initial-scale=1, maximum-scale=1

    <title>菜鸟教程(runoob.com)</title>
    <link href="http://www.runoob.com/static/ionic/css/ionic.min.cs
    <script src="http://www.runoob.com/static/ionic/js/ionic.bundle
  </head>
  <body ng-controller="AppCtrl">

    <ion-header-bar class="bar-positive">
      <h1 class="title">Contacts</h1>
      <div class="buttons">
        <button class="button button-icon ion-compose" ng-click="mo
        </button>
      </div>
    </ion-header-bar>
    <ion-content>
      <ion-list>
        <ion-item ng-repeat="contact in contacts">
          {{contact.name}}
        </ion-item>
      </ion-list>
    </ion-content>

    <script id="templates/modal.html" type="text/ng-template">
      <ion-modal-view>
        <ion-header-bar class="bar bar-header bar-positive">
          <h1 class="title">New Contact</h1>
          <button class="button button-clear button-primary" ng-cli
        </ion-header-bar>
        <ion-content class="padding">
```

```
            <div class="list">
              <label class="item item-input">
                <span class="input-label">First Name</span>
                <input ng-model="newUser.firstName" type="text">
              </label>
              <label class="item item-input">
                <span class="input-label">Last Name</span>
                <input ng-model="newUser.lastName" type="text">
              </label>
              <label class="item item-input">
                <span class="input-label">Email</span>
                <input ng-model="newUser.email" type="text">
              </label>
              <button class="button button-full button-positive" ng-c
            </div>
          </ion-content>
        </ion-modal-view>
      </script>

  </body>
</html>
```

CSS 代码

```
body {
  cursor: url('http://www.runoob.com/try/demo_source/finger.png'),
}
```

JavaScript 代码

```
angular.module('ionicApp', ['ionic'])

.controller('AppCtrl', function($scope, $ionicModal) {

  $scope.contacts = [
    { name: 'Gordon Freeman' },
    { name: 'Barney Calhoun' },
    { name: 'Lamarr the Headcrab' },
  ];

  $ionicModal.fromTemplateUrl('templates/modal.html', {
    scope: $scope
  }).then(function(modal) {
    $scope.modal = modal;
  });

  $scope.createContact = function(u) {
    $scope.contacts.push({ name: u.firstName + ' ' + u.lastName });
    $scope.modal.hide();
  };

});
```

ionic 导航

ion-nav-view

当用户在你的app中浏览时，ionic能够检测到浏览历史。通过检测浏览历史，实现向左或向右滑动时可以正确转换视图。

采用AngularUI路由器模块等应用程序接口可以分为不同的$state(状态)。Angular的核心为路由服务，URLs可以用来控制视图。

AngularUI路由提供了一个更强大的状态管理，即状态可以被命名，嵌套， 以及合并视图，允许一个以上模板呈现在同一个页面。

此外，每个状态无需绑定到一个URL，并且数据可以更灵活地推送到每个状态。

以下实例中，我们将创建一个应用程序中包含不同状态的导航视图。

我们的标记中选择ionNavView作为顶层指令。显示一个页眉栏我们用 ionNavBar 指令通过导航更新。

接下来，我们需要设置我们的将渲染的状态值。

```
var app = angular.module('myApp', ['ionic']);
app.config(function($stateProvider) {
  $stateProvider
  .state('index', {
    url: '/',
    templateUrl: 'home.html'
  })
  .state('music', {
    url: '/music',
    templateUrl: 'music.html'
  });
});
```

再打开应用，$stateProvider 会查询url, 看是否匹配 index 状态值, 再加载 index.html到<ion-nav-view>。

页面加载都是通过URLs配置的。在Angular中创建模板最一个简单的方式就是直接将他放到html模板文件中并且用

ionic 平台

$ionicPlatform

$ionicPlatform 用来检测当前的平台，以及诸如在PhoneGap/Cordova中覆盖Android后退按钮。

方法

```
onHardwareBackButton(callback)
```

有硬件的后退按钮的平台，可以用这种方法绑定到它。

`callback` : `function`

当该事件发生时，触发回调函数。

```
offHardwareBackButton(callback)
```

移除后退按钮的监听事件。

`callback` : `function`

最初绑定的监视器函数。

```
registerBackButtonAction(callback, priority, [actionId])
```

注册硬件后退按钮动作。当点击按钮时，只有一个动作会执行，因此该方法决定了注册的后退按钮动作具有最高的优先级。

例如，如果一个上拉菜单已经显示，后退按钮应该关闭上拉菜单，而不是返回一个页面视图或关闭一个打开的模型。

`callback` : `function`

当点击返回按钮时触发，如果该监视器具有最高的优先级。

`priority` : `number`

仅最高优先级的会执行。

`actionId(可选)` : `*`

该id指定这个动作。默认：一个随机且唯一的id。

返回值: 函数， 一个被触发的函数，将会注销 backButtonAction。

```
ready([callback])
```

设备准备就绪，则触发一个回调函数。

`callback(可选)` : `function=`

触发的函数。

返回: promise对象, 对象被构造 成功后得到解析。

ionic 浮动框

$ionicPopover

$ionicPopover 是一个可以浮在app内容上的一个视图框。

实例

HTML 代码

```
<p>
<button ng-click="openPopover($event)">打开浮动框</button>
</p>
<script id="my-popover.html" type="text/ng-template">
<ion-popover-view>
  <ion-header-bar>
    <h1 class="title">我的浮动框标题</h1>
  </ion-header-bar>
  <ion-content>
    Hello!
  </ion-content>
</ion-popover-view>
</script>
```

JavaScript 代码

```
angular.module('ionicApp', ['ionic'])
.controller( 'AppCtrl',['$scope','$ionicPopover','$timeout',functio

  $scope.popover = $ionicPopover.fromTemplateUrl('my-popover.html',
    scope: $scope
  });

  // .fromTemplateUrl() 方法
  $ionicPopover.fromTemplateUrl('my-popover.html', {
    scope: $scope
  }).then(function(popover) {
    $scope.popover = popover;
  });

  $scope.openPopover = function($event) {
    $scope.popover.show($event);
  };
  $scope.closePopover = function() {
    $scope.popover.hide();
  };
  // 清除浮动框
  $scope.$on('$destroy', function() {
    $scope.popover.remove();
  });
  // 在隐藏浮动框后执行
  $scope.$on('popover.hidden', function() {
    // 执行代码
  });
  // 移除浮动框后执行
  $scope.$on('popover.removed', function() {
    // 执行代码
  });

}])
```

ionic 对话框

$ionicPopup

ionic 对话框服务允许程序创建、显示弹出窗口。

$ionicPopup 提供了3个方法：alert(), prompt(),以及 confirm() 。

实例

HTML 代码

```
<body class="padding" ng-controller="PopupCtrl">
    <button class="button button-dark" ng-click="showPopup()">
      弹窗显示
    </button>
    <button class="button button-primary" ng-click="showConfirm()">
      确认对话框
    </button>
    <button class="button button-positive" ng-click="showAlert()">
      警告框
    </button>

    <script id="popup-template.html" type="text/ng-template">
      <input ng-model="data.wifi" type="text" placeholder="Passwor
    </script>
</body>
```

JavaScript 代码

```
angular.module('mySuperApp', ['ionic'])
.controller('PopupCtrl',function($scope, $ionicPopup, $timeout) {

 // Triggered on a button click, or some other target
 $scope.showPopup = function() {
   $scope.data = {}

   // 自定义弹窗
   var myPopup = $ionicPopup.show({
     template: '<input type="password" ng-model="data.wifi">',
     title: 'Enter Wi-Fi Password',
     subTitle: 'Please use normal things',
     scope: $scope,
```

```
      buttons: [
        { text: 'Cancel' },
        {
          text: '<b>Save</b>',
          type: 'button-positive',
          onTap: function(e) {
            if (!$scope.data.wifi) {
              // 不允许用户关闭，除非输入 wifi 密码
              e.preventDefault();
            } else {
              return $scope.data.wifi;
            }
          }
        },
      ]
    });
    myPopup.then(function(res) {
      console.log('Tapped!', res);
    });
    $timeout(function() {
       myPopup.close(); // 3秒后关闭弹窗
    }, 3000);
   };
    //  confirm 对话框
    $scope.showConfirm = function() {
      var confirmPopup = $ionicPopup.confirm({
        title: 'Consume Ice Cream',
        template: 'Are you sure you want to eat this ice cream?'
      });
      confirmPopup.then(function(res) {
        if(res) {
          console.log('You are sure');
        } else {
          console.log('You are not sure');
        }
      });
    };

    //  alert（警告） 对话框
    $scope.showAlert = function() {
      var alertPopup = $ionicPopup.alert({
        title: 'Don\'t eat that!',
        template: 'It might taste good'
      });
      alertPopup.then(function(res) {
        console.log('Thank you for not eating my delicious ice crear
      });
    };
 });
```

ionic 滚动条

ion-scroll

ion-scroll 用于创建一个可滚动的容器。

用法

```
<ion-scroll
    [delegate-handle=""]
    [direction=""]
    [paging=""]
    [on-refresh=""]
    [on-scroll=""]
    [scrollbar-x=""]
    [scrollbar-y=""]
    [zooming=""]
    [min-zoom=""]
    [max-zoom=""]>
    ...
</ion-scroll>
```

API

`delegate-handle(可选)` ： `字符串`

该句柄利用 `$ionicScrollDelegate` 指定滚动视图。

`direction(可选)` ： `字符串`

滚动的方向。 'x' 或 'y'。 默认 'y'。

`paging(可选)` ： `布尔值`

分页是否滚动。

`on-refresh(可选)` ： `表达式`

调用下拉刷新， 由 `ionRefresher` 触发。

`on-scroll(可选)` ： `表达式`

当用户滚动时触发。

`scrollbar-x(可选)` ： `布尔值`

是否显示水平滚动条。默认为false。

`scrollbar-y(可选)` : `布尔值`

是否显示垂直滚动条。默认为true。

`zooming(可选)` : `布尔值`

是否支持双指缩放。

`min-zoom(可选)` : `整数`

允许的最小缩放量（默认为0.5）

`max-zoom(可选)` : `整数`

允许的最大缩放量（默认为3）

实例

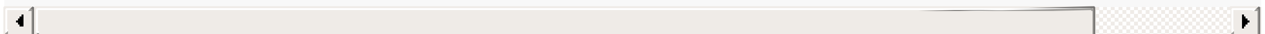
HTML 代码

```
<ion-scroll zooming="true" direction="xy" style="width: 500px; heig
    <div style="width: 5000px; height: 5000px; background: url('htt
</ion-scroll>
```



CSS 代码

```
body {
  cursor: url('http://www.runoob.com/try/demo_source/finger.png'),
}
```



JavaScript 代码

```
angular.module('ionicApp', ['ionic']);
```

ion-infinite-scroll

当用户到达页脚或页脚附近时，ionInfiniteScroll指令允许你调用一个函数 。

当用户滚动的距离超出底部的内容时，就会触发你指定的on-infinite。

用法

```
<ion-content ng-controller="MyController">
  <ion-infinite-scroll
    on-infinite="loadMore()"
    distance="1%">
  </ion-infinite-scroll>
</ion-content>
```

```
function MyController($scope, $http) {
  $scope.items = [];
  $scope.loadMore = function() {
    $http.get('/more-items').success(function(items) {
      useItems(items);
      $scope.$broadcast('scroll.infiniteScrollComplete');
    });
  };

  $scope.$on('stateChangeSuccess', function() {
    $scope.loadMore();
  });
}
```

当没有更多数据加载时，就可以用一个简单的方法阻止无限滚动，那就是angular的ng-if 指令:

```
<ion-infinite-scroll
  ng-if="moreDataCanBeLoaded()"
  icon="ion-loading-c"
  on-infinite="loadMoreData()">
</ion-infinite-scroll>
```

API

`on-infinite` : `表达式`

当滚动到底部时触发的时间。

`distance(可选)` : `字符串`

从底部滚动到触发on-infinite表达式的距离。默认: 1%。

`icon(可选)` : `字符串`

当加载时显示的图标。默认: 'ion-loading-d'。

$ionicScrollDelegate

授权控制滚动视图（通过ion-content 和 ion-scroll指令创建）。

该方法直接被$ionicScrollDelegate服务触发，来控制所有滚动视图。用$getByHandle方法控制特定的滚动视图。

用法

```
<body ng-controller="MainCtrl">
  <ion-content>
    <button ng-click="scrollTop()">滚动到顶部!</button>
  </ion-content>
</body>
```

```
function MainCtrl($scope, $ionicScrollDelegate) {
  $scope.scrollTop = function() {
    $ionicScrollDelegate.scrollTop();
  };
}
```

方法

```
resize()
```

告诉滚动视图重新计算它的容器大小。

```
scrollTop([shouldAnimate])
```

`shouldAnimate(可选)` ： `布尔值`

是否应用滚动动画。

```
scrollBottom([shouldAnimate])
```

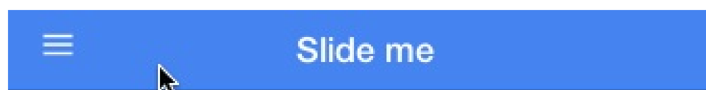
`shouldAnimate(可选)` ： `布尔值`

是否应用滚动动画。

ionic 侧栏菜单

一个容器元素包含侧边菜单和主要内容。通过把主要内容区域从一边拖动到另一边，来让左侧或右侧的侧栏菜单进行切换。

效果图如下所示：



用法

要使用侧栏菜单，添加一个父元素<ion-side-menus>，一个中间内容 <ion-side-menu-content>，和一个或更多 <ion-side-menu> 指令。

```
<ion-side-menus>
  <!-- 中间内容 -->
  <ion-side-menu-content ng-controller="ContentController">
  </ion-side-menu-content>

  <!-- 左侧菜单 -->
  <ion-side-menu side="left">
  </ion-side-menu>

  <!-- 右侧菜单 -->
  <ion-side-menu side="right">
  </ion-side-menu>
</ion-side-menus>
```

```
function ContentController($scope, $ionicSideMenuDelegate) {
  $scope.toggleLeft = function() {
    $ionicSideMenuDelegate.toggleLeft();
  };
}
```

API

`enable-menu-with-back-views(可选)` : `布尔值`

在返回按钮显示时，确认是否启用侧边栏菜单。

`delegate-handle` : 字符串 该句柄用于标识带有$ionicScrollDelegate的滚动视图。

ion-side-menu-content

一个可见主体内容的容器，同级的一个或多个ionSideMenu 指令。

用法

```
<ion-side-menu-content
  drag-content="true">
</ion-side-menu-content>
```

API

`drag-content(可选)` : `布尔值`

内容是否可被拖动。默认为true。

ion-side-menu

一个侧栏菜单的容器，同级的一个ion-side-menu-content 指令。

用法

```
<ion-side-menu
  side="left"
  width="myWidthValue + 20"
  is-enabled="shouldLeftSideMenuBeEnabled()">
</ion-side-menu>
```

API

`side` : `字符串`

侧栏菜单当前在哪一边。可选的值有: 'left' 或 'right'。

`is-enabled(可选)` : `布尔值`

该侧栏菜单是否可用。

`width(可选)` : `数值`

侧栏菜单应该有多少像素的宽度。默认为275。

menu-toggle

在一个指定的侧栏中切换菜单。

用法

下面是一个在导航栏内链接的例子。点击此链接会自动打开指定的侧栏菜单。

```
<ion-view>
  <ion-nav-buttons side="left">
   <button menu-toggle="left" class="button button-icon icon ion-na
  </ion-nav-buttons>
 ...
</ion-view>
```

menu-close

关闭当前打开的侧栏菜单。

用法

下面是一个在导航栏内链接的例子。点击此链接会自动打开指定的侧栏菜单。

```
<a menu-close href="#/home" class="item">首页</a>
```

$ionicSideMenuDelegate

该方法直接触发$ionicSideMenuDelegate服务，来控制所有侧栏菜单。用$getByHandle方法控制特定情况下的ionSideMenus。

用法

```
<body ng-controller="MainCtrl">
  <ion-side-menus>
    <ion-side-menu-content>
      内容！
      <button ng-click="toggleLeftSideMenu()">
        切换左侧侧栏菜单
      </button>
    </ion-side-menu-content>
    <ion-side-menu side="left">
      左侧菜单！
    <ion-side-menu>
  </ion-side-menus>
</body>
```

```
function MainCtrl($scope, $ionicSideMenuDelegate) {
  $scope.toggleLeftSideMenu = function() {
    $ionicSideMenuDelegate.toggleLeft();
  };
}
```

方法

```
toggleLeft([isOpen])
```

切换左侧侧栏菜单（如果存在）。

`isOpen(可选)` : `布尔值`

是否打开或关闭菜单。默认：切换菜单。

```
toggleRight([isOpen])
```

切换右侧侧栏菜单（如果存在）。

`isOpen(可选)` ： `布尔值`

是否打开或关闭菜单。默认：切换菜单。

```
getOpenRatio()
```

获取打开菜单内容超出菜单宽度的比例。比如，一个宽度为100px的菜单被宽度为50px以50%的比例打开，将会返回一个比例值为0.5。

返回值: 浮点 0 表示没被打开，如果左侧菜单处于已打开或正在打开为0 到 1，如果右侧菜单处于已打开或正在打开为0 到-1。

```
isOpen()
```

返回值: 布尔值，判断左侧或右侧菜单是否已经打开。

```
isOpenLeft()
```

返回值: 布尔值左侧菜单是否已经打开。

```
isOpenRight()
```

返回值: 布尔值右侧菜单是否已经打开。

```
canDragContent([canDrag])
```

`canDrag(可选)` ： `布尔值`

设置是否可以拖动内容打开侧栏菜单。

返回值: 布尔值，是否可以拖动内容打开侧栏菜单。

```
$getByHandle(handle)
```

`handle` ： `字符串`

例如:

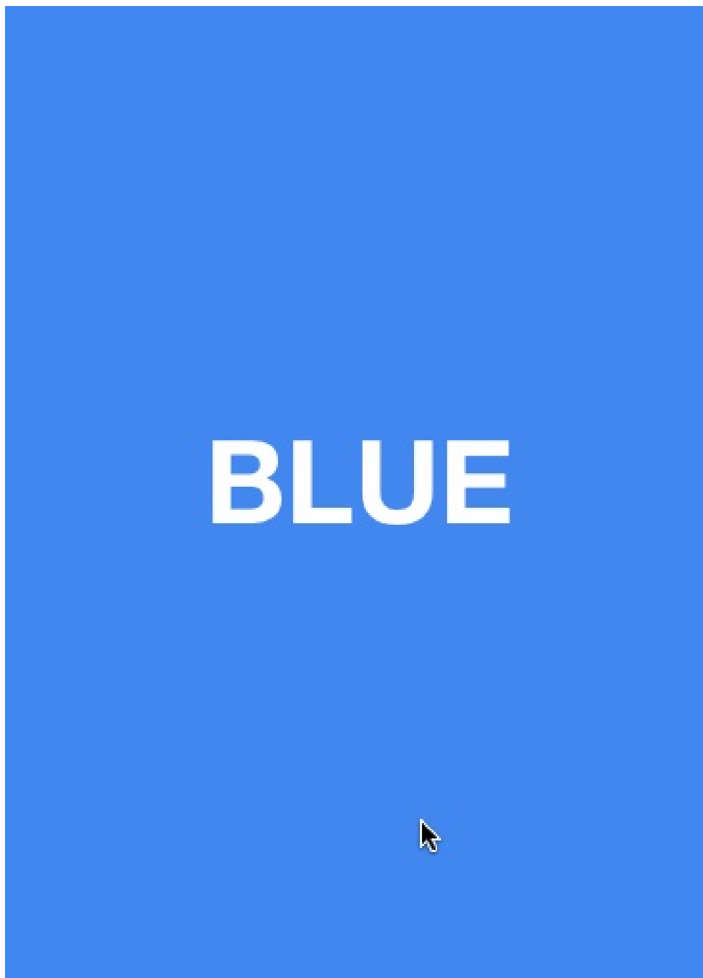
```
$ionicSideMenuDelegate.$getByHandle('my-handle').toggleLeft();
```

ionic 滑动框

ion-slide-box

滑动框是一个包含多页容器的组件，每页滑动或拖动切换：

效果图如下：



用法

```
<ion-slide-box on-slide-changed="slideHasChanged($index)">
  <ion-slide>
    <div class="box blue"><h1>BLUE</h1></div>
  </ion-slide>
  <ion-slide>
    <div class="box yellow"><h1>YELLOW</h1></div>
  </ion-slide>
  <ion-slide>
    <div class="box pink"><h1>PINK</h1></div>
  </ion-slide>
</ion-slide-box>
```

API

`delegate-handle(可选)` : `字符串`

该句柄用 `$ionicSlideBoxDelegate` 来标识这个滑动框。

`does-continue(可选)` : `布尔值`

滑动框是否自动滑动。

`slide-interval(可选)` : `数字`

等待多少毫秒开始滑动（如果继续则为true）。默认为4000。

`show-pager(可选)` : `布尔值`

滑动框的页面是否显示。

`pager-click(可选)` : `表达式`

当点击页面时，触发该表达式（如果shou-pager为true）。传递一个'索引'变量。

`on-slide-changed(可选)` : `表达式`

当滑动时，触发该表达式。传递一个'索引'变量。

`active-slide(可选)` : `表达式`

将模型绑定到当前滑动框。

实例

HTML 代码

```
<ion-slide-box active-slide="myActiveSlide">
    <ion-slide>
      <div class="box blue"><h1>BLUE</h1></div>
    </ion-slide>
    <ion-slide>
      <div class="box yellow"><h1>YELLOW</h1></div>
    </ion-slide>
    <ion-slide>
      <div class="box pink"><h1>PINK</h1></div>
    </ion-slide>
</ion-slide-box>
```

CSS 代码

```
.slider {
  height: 100%;
}
.slider-slide {
  padding-top: 80px;
  color: #000;
  background-color: #fff;
  text-align: center;

  font-family: "HelveticaNeue-Light", "Helvetica Neue Light", "Hel
  font-weight: 300;
}

.blue {
  background-color: blue;
}

.yellow {
  background-color: yellow;
}

.pink {
  background-color: pink;
}
```

JavaScript 代码

```
angular.module('ionicApp', ['ionic'])

.controller('SlideController', function($scope) {

  $scope.myActiveSlide = 1;

})
```

ionic 加载动画

ion-spinner

ionSpinner 提供了许多种旋转加载的动画图标。当你的界面加载时，你就可以呈现给用户相应的加载图标。

该图标采用的是SVG。

用法

```
<ion-spinner icon="spiral"></ion-spinner>    //默认用法
```

像大部分其他的ionic组件一样，spinner也可以使用ionic的标准颜色命名规则，就像下面这样：

```
<ion-spinner class="spinner-energized"></ion-spinner>
```

实例

HTML 代码

```
<ion-content scroll="false" class="has-header">
  <p>
    <ion-spinner icon="android"></ion-spinner>
    <ion-spinner icon="ios"></ion-spinner>
    <ion-spinner icon="ios-small"></ion-spinner>
    <ion-spinner icon="bubbles" class="spinner-balanced"></ion-spir
    <ion-spinner icon="circles" class="spinner-energized"></ion-sp
  </p>

  <p>
    <ion-spinner icon="crescent" class="spinner-royal"></ion-spinne

    <ion-spinner icon="dots" class="spinner-dark"></ion-spinner>
    <ion-spinner icon="lines" class="spinner-calm"></ion-spinner>
    <ion-spinner icon="ripple" class="spinner-assertive"></ion-spir
    <ion-spinner icon="spiral"></ion-spinner>
  </p>

</ion-content>
```

CSS 代码

```
body {
  cursor: url('http://www.runob.com/try/demo_source/finger.png'), a
}
p {
  text-align: center;
  margin-bottom: 40px !important;
}
.spinner svg {
  width: 19% !important;
  height: 85px !important;
}
```

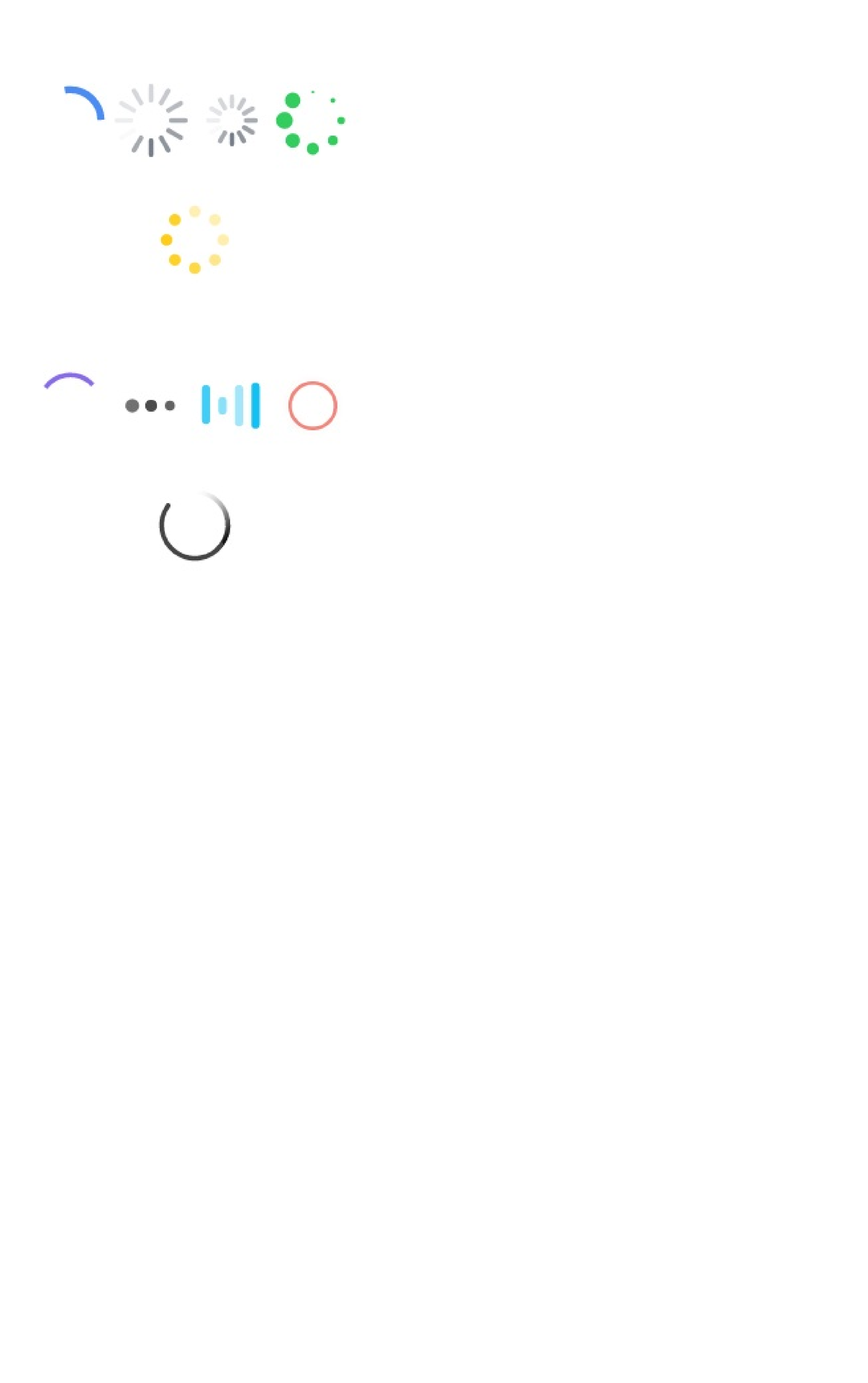
JavaScript 代码

```
angular.module('ionicApp', ['ionic'])

.controller('MyCtrl', function($scope) {

});
```

效果如下所示：



ionic 选项卡栏操作

ion-tabs

ion-tabs 是有一组页面选项卡组成的选项卡栏。可以通过点击选项来切换页面。

对于 iOS，它会出现在屏幕的底部，Android会出现在屏幕的顶部(导航栏下面)。

用法

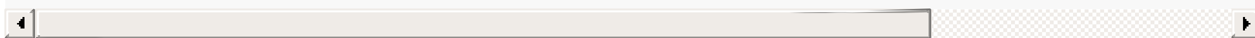
```
<ion-tabs class="tabs-positive tabs-icon-only">

  <ion-tab title="首页" icon-on="ion-ios7-filing" icon-off="ion-ios
    <!-- 标签 1 内容 -->
  </ion-tab>

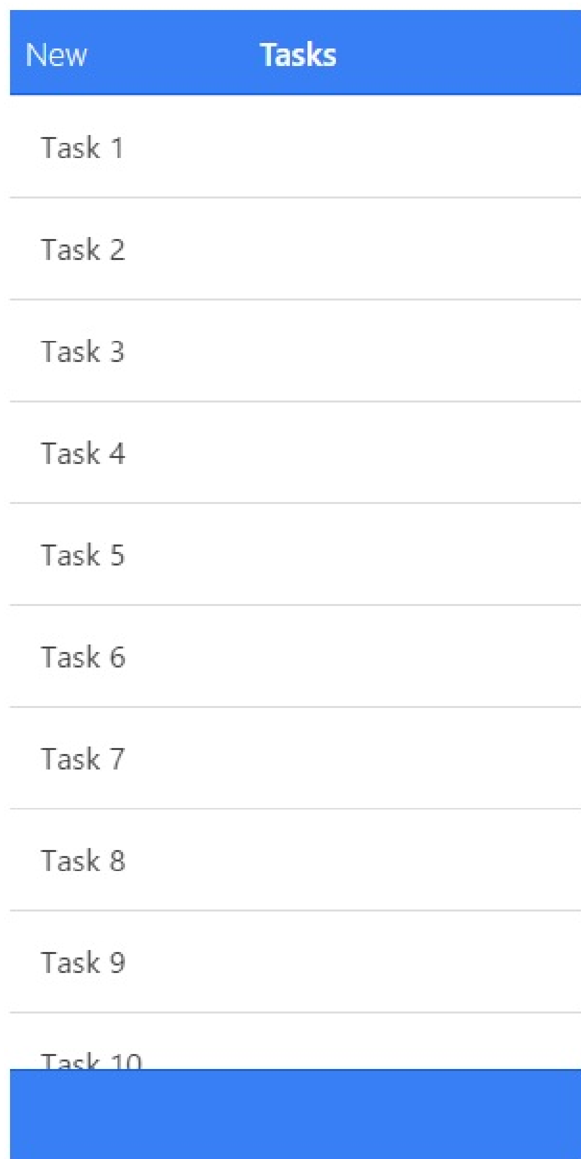
  <ion-tab title="关于" icon-on="ion-ios7-clock" icon-off="ion-ios7
    <!-- 标签 2 内容 -->
  </ion-tab>

  <ion-tab title="设置" icon-on="ion-ios7-gear" icon-off="ion-ios7-
    <!-- 标签 3 内容 -->
  </ion-tab>

</ion-tabs>
```



效果如下所示：



API

`delegate-handle(可选)` : `字符串`

该句柄用 `$ionicTabsDelegate` 来标识这些选项卡。

ion-tab

隶属于ionTabs

包含一个选项卡内容。该内容仅存在于被选中的给定选项卡中。

每个ionTab都有自己的浏览历史。

用法

```
<ion-tab
  title="Tab!"
  icon="my-icon"
  href="#/tab/tab-link"
  on-select="onTabSelected()"
  on-deselect="onTabDeselected()">
</ion-tab>
```

API

`title` ： `字符串`

选项卡的标题。

`href(可选)` ： `字符串`

但触碰的时候，该选项卡将会跳转的的链接。

`icon(可选)` ： `字符串`

选项卡的图标。如果给定值，它将成为ion-on和ion-off的默认值。

`icon-on(可选)` ： `字符串`

被选中标签的图标。

`icon-off(可选)` ： `字符串`

没被选中标签的图标。

`badge(可选)` ： `表达式`

选项卡上的徽章（通常是一个数字）。

`badge-style(可选)` ： `表达式`

选项卡上微章的样式（例，tabs-positive ）。

`on-select(可选)` ： `表达式`

选项卡被选中时触发。

`on-deselect(可选)` ： `表达式`

选项卡取消选中时触发。

`ng-click(可选)` ： `表达式`

通常，点击时选项卡会被选中。如果设置了 ng-Click，它将不会被选中。 你可以用$ionicTabsDelegate.select()来指定切换标签。

$ionicTabsDelegate

授权控制ionTabs指令。

该方法直接调用$ionicTabsDelegate服务，控制所有ionTabs指令。用$getByHandle方法控制具体的ionTabs实例。

用法

```
<body ng-controller="MyCtrl">
  <ion-tabs>

    <ion-tab title="Tab 1">
      你好，标签1！
      <button ng-click="selectTabWithIndex(1)">选择标签2</button>
    </ion-tab>
    <ion-tab title="Tab 2">你好标签2！</ion-tab>

  </ion-tabs>
</body>
```

```
function MyCtrl($scope, $ionicTabsDelegate) {
  $scope.selectTabWithIndex = function(index) {
    $ionicTabsDelegate.select(index);
  }
}
```

方法

```
select(index, [shouldChangeHistory])
```

选择标签来匹配给定的索引。

`index` ： `数值`

选择标签的索引。

`shouldChangeHistory(可选)` ： `布尔值`

此选项是否应该加载这个标签的浏览历史（如果存在），并使用，或仅加载默认页面。默认为false。提示：如果一个 `ion-nav-view` 在选项卡里，你可能需要设置它为true。

```
selectedIndex()
```

返回值: 数值, 被选中标签的索引，如 -1。

```
$getByHandle(handle)
```

`handle` ： `字符串`

例如:

```
$ionicTabsDelegate.$getByHandle('my-handle').select(0);
```

免责声明

版权信息

菜鸟教程（www.runoob.com）刊载的所有内容，包括文字、图片、音频、视频、软件、程序、以及网页版式设计等均在网上搜集。

菜鸟教程提供的内容仅用于个人学习、研究或欣赏。我们不保证内容的正确性。通过使用本站内容随之而来的风险与本站无关

访问者可将本网站提供的内容或服务用于个人学习、研究或欣赏，以及其他非商业性或非盈利性用途，但同时应遵守著作权法及其他相关法律的规定，不得侵犯本网站及相关权利人的合法权利。

本网站内容原作者如不愿意在本网站刊登内容，请及时通知本站，予以删除。

本网站的编程技术内容大部分翻译自 W3Schools Online Web Tutorials 与 Tutorialspoint, 内容原作者如不愿意在本网站刊登内容，请及时通知本站，予以删除。

链接到菜鸟教程

任何网站都可以链接到菜鸟教程的任何页面。

如果您需要在对少量内容进行引用，请务必在引用该内容的页面添加指向被引用页面的链接。

保证

菜鸟教程不提供任何形式的保证。所有与使用本站相关的直接风险均由用户承担。菜鸟教程提供的所有代码均为实例，并不对性能、适用性、适销性或/及其他方面提供任何保证。

菜鸟教程的内容可能包含不准确性或错误。菜鸟教程不对本网站及其内容的准确性进行保证。如果您发现本站点及其内容包含错误，请联系我们以便这些错误得到及时的更正：429240967@qq.com

您的行为

当您使用本站点时，说明您已经同意并接受本页面的所有信息。

联系方式

联系邮箱：429240967@qq.com（相关事务请发函至该邮箱）